



MONTCLAIR STATE
UNIVERSITY

Montclair State University
**Montclair State University Digital
Commons**

Theses, Dissertations and Culminating Projects

5-2007

Parallel Nonnegative Matrix Factorization Algorithms for Hyperspectral Images

Lukasz Grzegorz Maciak

Follow this and additional works at: <https://digitalcommons.montclair.edu/etd>



Part of the [Computer Sciences Commons](#)

Parallel Nonnegative Matrix Factorization Algorithms for Hyperspectral Images

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Masters in Computer Science

by

Lukasz Grzegorz Maciak
Montclair State University
Montclair, NJ

2007

Abstract

Hyperspectral imaging is a branch of remote sensing which deals with creating and processing aerial or satellite pictures that capture wide range of wavelengths, most of which are invisible to the naked eye. Hyperspectral images are composed of many bands, each corresponding to certain light frequencies. Because of their complex nature, image processing tasks such as feature extraction can be resource and time consuming. There are many unsupervised extraction methods available. A recently investigated one is Nonnegative Matrix Factorization (NMF), a method that given positive linear matrix of positive sources, attempts to recover them. In this thesis we designed, implemented and tested parallel versions of two popular iterative NMF algorithms: one based on multiplicative updates, and another on alternative gradient computation.

Our algorithms are designed to leverage the multi-processor SMP architecture and power of threading to evenly distribute the workload among the available CPU's and improve the performance as compared to their sequential counterparts. This work could be used as a basis for creating even more powerful distributed algorithms that would work on clustered architectures. The experiments show a speedup in both algorithms without reduction in accuracy.

In addition, we have also developed a java based framework offering reading and writing tools for various hyperspectral image types, as well as visualization tools, and a graphical user interface to launch and control the factorization processes.

MONTCLAIR STATE UNIVERSITY

Parallel Nonnegative Matrix Factorization Algorithms for
Hyperspectral Images

by

Lukasz Grzegorz Maciak

A Master's Thesis Submitted to the Faculty of
Montclair State University

In Partial Fulfillment of the Requirements

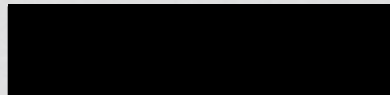
For the Degree of
Master of Computer Science

May, 2007


College of Science and Mathematics

Thesis Committee:

Department of Computer Science

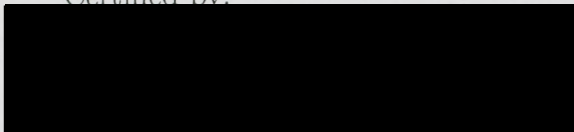
 4/24/07

Thesis Sponsor: Dr. Stefan A. Robila

 4/24/07

Certified by:

Committee Member: Dr. Angel Gutierrez

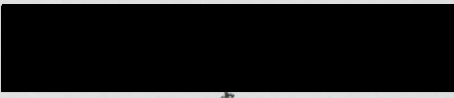


 4/24/07

Dean of College or School

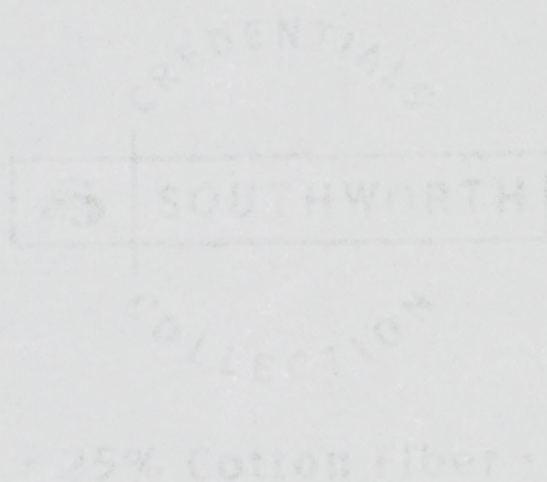
Committee Member: Dr. John Jenq

4/24/07
Date

 4/24/07

Department Chair: Dr. Dorothy Deremer

Copyright © 2007 by *Lukasz Grzegorz Maciak*. All rights reserved.




RESISTANCE

COLLECTION

• 25% Cotton Fiber •

RESISTANCE

 SOUTHWORTH

Acknowledgments

The completion of this thesis would not be possible if it was not for the help and support of the Computer Science Department faculty and staff. I would like to express my deepest thanks to all the people who helped and supported me or somehow contributed to this thesis.

I would like to extend special thanks to:

Dr. Stefan Robila, my thesis adviser, for always being supportive and accommodating and ready to answer all my questions.

Dr. Deremer for continuous support throughout my graduate career.

Dr. Gutierrez and Dr. Jenq who graciously agreed to be my thesis committee.

Dr. Koeller who introduced me to LaTeX and made the process of writing and compiling the thesis much easier.

Beverly Macaluso for help and assistance with all the clerical and administrative matters during my graduate studies.

All the Computer Science faculty members who kept me motivated and helped me to succeed over the years.

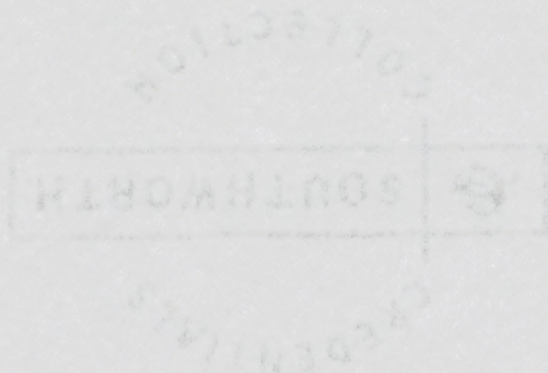
Work on this project has been supported by equipment awarded under the Sun Microsystems Academic Excellence Grant #EDUD-7824-060154-US, Stefan Robila PI.

Contents

Acknowledgments	ii
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Remote Sensing and Hyperspectral Data	3
1.2 Challenges in Hyperspectral Data Processing	4
1.2.1 Visualization	5
1.2.2 I/O and Encoding	7
1.3 Feature Extraction	9
1.3.1 Dimensionality Reduction	9
1.3.2 Linear Mixing Model (LMM)	10
1.3.3 Supervised Feature Extraction	12
1.3.4 Unsupervised Feature Extraction	13
2 Nonnegative Matrix Factorization Methods	15
2.1 Nonnegative Matrix Factorization (NMF)	15
2.1.1 NMF: General Problem Description	15
2.1.2 Applying NMF to Hyperspectral Data	18
2.1.3 Standard Iterative NMF Algorithm	19
2.1.4 Projected Gradient NMF (PG-NMF)	21
2.1.5 Other NMF Approaches	25
2.1.5.1 Stochastic NMF (S-NMF)	25
2.1.5.2 Multilayer NMF	26
2.1.5.3 PG-NMF with Initial Newton Step	27
2.1.6 Convergence Criteria	28
2.2 Other Methods	30
2.2.1 Principal Component Analysis	31
2.2.2 Independent Component Analysis	33
2.2.3 Linear Mixing Model and PCA/ICA Feature Extraction	34

3	Parallel Approaches and NMF Algorithms	36
3.1	Different Parallel Approaches	36
3.1.1	The Need for Parallel Implementation of NMF	36
3.1.2	Overview of Parallel Architectures	37
3.2	Data Distribution	39
3.2.1	Evaluation Criteria: Speedup	40
3.3	Previous Efforts	41
3.4	Parallel NMF Algorithms	42
3.4.1	Implementation Details	42
3.4.2	Parallel Nonnegative Matrix Factorization (P-NMF)	43
3.4.3	Parallel Projected Gradient Nonnegative Matrix Factorization (PPG-NMF)	46
4	Experimental Results	50
4.1	Experimental Data Set	50
4.2	Testing Platform and Machine Limitations	53
4.3	Testing Procedure	54
4.4	Results	54
4.4.1	P-NMF	55
4.4.2	PPG-NMF	60
4.4.3	Discussion of Results	66
5	Java Based Hyperspectral Image Processing Toolkit	67
5.1	Introduction	67
5.2	Data Structures and Representation	68
5.3	Designing a Graphical User Interface	69
5.3.1	Choosing a Widget Toolkit	69
5.3.2	GUI Overview	71
5.3.3	Visualization of Hyperspectral Data	75
5.3.4	Starting the GUI Interface	77
5.4	Handling I/O	77
5.4.1	Unified Framework for Handling Different Hyperspectral Data Types	78
5.4.2	Big Endian/Little Endian Conversion	79
5.5	Command Line User Interface	81
6	Conclusions and Future Work	84
	Bibliography	85
	Appendices	90
A	Source Code	91
A.1	Documentation	92
A.2	Compiling the Code	92
A.3	Running The Code	93

A.4	Extending the Toolkit	93
A.4.1	Adding new Reader or Writer Class	93
A.4.2	Adding new Factorization Algorithm	93
A.4.3	Adding new DataBuffer	94
A.5	Known Issues	94



List of Figures

1.1	Conceptual Model of a Hyperspectral Image	5
1.2	Visualization of a Hyperspectral Image	6
1.3	Different Encoding Methods	8
1.4	Reduction of Data Dimensionality	10
1.5	Linear Mixing Model	11
2.1	Visual Example of Non Negative Matrix Factorization	16
2.2	Nonnegative Matrix Factorization	18
2.3	Hyperspectral Data and NMF	19
2.4	NMF Algorithm	20
2.5	PG-NMF Algorithm	24
2.6	Sample Graph of $f(W,H)$	29
3.1	Data Distribution in P-NMF	43
3.2	Diagram of P-NMF with 4 Threads	45
3.3	PPG-NMF Diagram	49
4.1	Experimental Data	51
4.2	HYDICE Data Sample with the panels hilighted	51
4.3	Abundance Graphs for different materials present in HYDICE sample	52
4.4	Abundance Graphs for different materials present in SOC700 sample	53
4.5	Accuracy Graph for P-NMF applied to the HYDICE data sample . .	56
4.6	Accuracy Graph for P-NMF applied to the SOC 700 data sample . .	56
4.7	Average Execution Time for P-NMF applied to the HYDICE data sample	57
4.8	Average Execution Time for P-NMF applied to the SOC 700 data sample	57
4.9	Average Time per Iteration for P-NMF applied to the HYDICE data sample	58
4.10	Average Time per Iteration for P-NMF applied to the SOC 700 data sample	58
4.11	Average Speedup for P-NMF applied to the HYDICE data sample . .	59
4.12	Average Speedup for P-NMF applied to the SOC 700 data sample . .	59
4.13	Accuracy Graph for PPG-NMF applied to the HYDICE data sample	61
4.14	Average Execution Time PPG-NMF applied to the HYDICE data sample	61
4.15	Average Execution Time per Iteration PPG-NMF applied to the HY- DICE data sample	62

4.16	Speedup for PPG-NMF applied to the HYDICE data sample	62
4.17	Accuracy Graph for PPG-NMF applied to the SOC 700 sample	63
4.18	Average Execution Time for PPG-NMF applied to the SOC 700 sample	63
4.19	Average Execution Time per Iteration for PPG-NMF applied to the SOC 700 sample	64
4.20	Speedup for PPG-NMF applied to the SOC 700 sample	64
4.21	Rounds Per Iteration in PPG-NMF (SOC 700 sample)	65
5.1	HyperJ GUI: Main Window	72
5.2	HyperJ GUI: Open File Dialog	72
5.3	HyperJ GUI: Open File Dialog Details	73
5.4	HyperJ GUI: Image Toolbox	74
5.5	HyperJ GUI: HyperJ with an open image and text dump	74
5.6	Visualization of Hyperspectral Images using Java	76
5.7	Partial Class Diagram of the I/O Framework	80

List of Tables

5.1 CLI Arguments	82
-----------------------------	----

Chapter 1

Introduction

In this paper we describe, implement and test a fast, unsupervised, parallel feature extraction algorithms for hyperspectral images using Nonnegative Matrix Factorization (NMF). The implementation leverages two different distributed processing paradigms in order to speed up the time consuming processing stage of the algorithm. It is developed as a part of a self contained hyperspectral imaging framework which includes visualization tools, advanced I/O and conversion tools, user friendly graphical user interface, and logging tools.

We looked at two prominent feature extraction algorithms: the original NMF algorithm as defined by Lee and Seung [1] and the projected gradient optimization described by Chih-Jen Lin [2]. We developed parallel versions of these two algorithms which are semantically identical to the originals, but do perform faster on multi CPU machines.

All the code was written in Java in order to take full benefit of the language's platform independence and rich distributed processing features. The aim was to minimize the number of software and hardware dependencies for the working program, and streamline the build procedure and installation. Thanks to Java's compile-once,

run anywhere feature the code can be rapidly deployed on various machines provided that they are able to host Java Virtual Machine. The finalized set of binaries can be distributed as a single compressed jar file that can be used to invoke the graphical user interface, or run heavy duty processing directly from the command line.

Our implementation used threading for parallel execution and is designed to run on SMP architectures. Java has very good native support for this type of parallel programming. Because of this our code is not dependent on any third party libraries, or platform specific code adding much flexibility to our solution. In addition to threading, we also briefly discuss other alternatives such as remote procedure calls but we do not provide code samples or test results for these methods.

Due to their size and organization, hyperspectral images are usually not manageable by mainstream image viewing toolkits. Thus visualization and I/O capabilities were an important part of the project. We have developed an object oriented, modular toolkit that can be easily extended to accommodate new or non-standard file formats.

We will start this paper by explaining the nature of Remote Sensing, and talk about Hyperspectral images and their features. Then we will describe challenges in implementation an automated system to process these images. We will provide an explanation of what is feature extraction, and describe both of the featured NMF algorithms in detail. Finally we will go over the proposed parallelization strategies for our algorithm, test their performance and discuss the results. We will also provide detailed description of the implemented features in the Java based hyperspectral processing toolkit we produced.

1.1 Remote Sensing and Hyperspectral Data

The study of Earth's surface and atmospheric features from a distance is usually referred to as remote sensing [3]. It is a branch of science related closely to aerospace technology and has applications in many fields including agriculture, forestry, meteorology and military surveillance. It involves direct observation, reconnaissance and aerial or satellite photography and video. Thanks to recent advances in optics, the focus shifted almost entirely toward photography and video. One of the most important features of photography is that an optical lens can usually register much more information than a human eye. Using specialized sensory equipment it is possible to take a picture which will include the full spectrum of reflected light, and not the only the tiny fraction that is visible to humans [4].

Hyperspectral sensors are usually able to record reflected light frequencies between $0.4\mu m$ and $2.4\mu m$. Only wavelengths from $0.4\mu m$ to $0.7\mu m$ are visible for a naked eye [5]. Data is collected as hundreds of images, each corresponding to a narrow wavelength interval. For example, the AVIRIS sensing system used by NASA's Jet Propulsion Laboratory, simultaneously takes 244 images, each with spectral resolution of $10nm$ (or $0.01\mu m$) (See Figure 1.1) [6]. Each individual image is called a band.

A hyperspectral image is therefore a three dimensional data structure (image cube), in which each pixel can be addressed by using 3 coordinates x , y and z representing vertical position, horizontal position, and band number respectively. All the pixels with the same x and y coordinates, are closely related as they represent the same point in space. Collection of these related pixels can be referred to as pixel vector. This vector can be graphed as a continuous function of the wavelengths. Figure 1.1 shows some sample graphs for selected ground features. The function of a pixel vector is known as a spectra. Different photographed materials will have unique spectra, due to the differences in reflectance at different wavelengths.

SECTION

SOUTH WORTH

CREDENTIALS

Hyperspectral images can convey wealth of information, but also have important drawbacks. Very often, the photographed area may include unknown targets that are smaller than the ground sampling distance (GSD). These end up embedded within a single pixel and cannot be identified using visual inspection. Their presence can only be inferred by analyzing their spectral properties. Thus, traditional spatial based methods used for image analysis cannot be used for hyperspectral data which requires strong focus on finding, identifying and classifying sub-pixel targets [7].

Hyperspectral data samples can often take up anywhere from tens to hundreds of megabytes. For example, a 640 by 640 image, with 240 bands may take up 96 megabytes, if we assume that each pixel is represented by a single byte. During processing it might be necessary to use floating point notation in which case the space required to store the image quadruples [5]. The complexity of all processing algorithms, is inevitably tied to the size of the sample, and therefore to number of spectral bands. It is desirable to reduce the number of spectral dimensions as much as possible before performing any heavy duty analysis. Since it can be assumed that some spectral bands contain very little but noise or redundant data, this reduction can be accomplished without significant loss of information. The process of reducing a hyperspectral sample dimensionally by detecting and isolating meaningful features of the image is known as feature extraction.

1.2 Challenges in Hyperspectral Data Processing

Due to their complexity, handling hyperspectral images poses many challenges to the researchers. Multi-band data is difficult to visualize and store in a meaningful way. Furthermore, they often include large amounts of additive noise that needs to be reduced using different feature extraction methods.

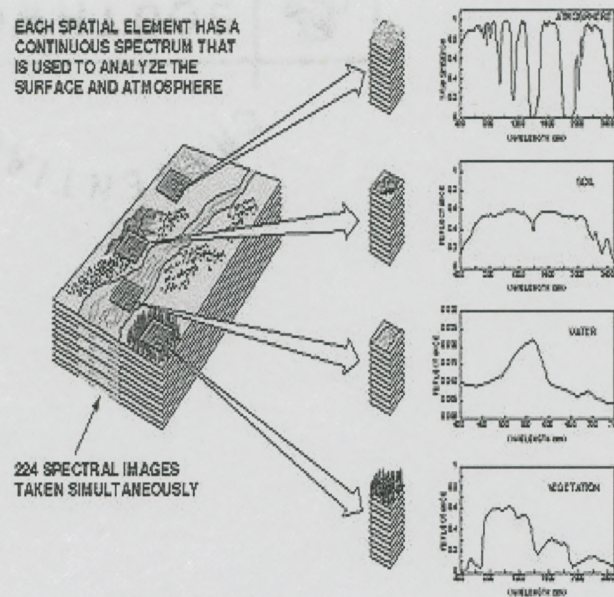


Figure 1.1: Conceptual Model of a Hyperspectral Image [8]

1.2.1 Visualization

The frequency range of the light captured by remote sensing devices far grater than that which can be registered by the human eye. It is impossible to simultaneously visualize all the bands of a hypierspectral image in a way that would be accessible to a human being. However it is possible to break down the image into distinct components that can be displayed and printed as standard 2d images. This way researchers can examine both the data represented in the visible spectrum, as well as information, contained in some of the invisible bands. There are two distinct ways to partially visualize hyperspectral data, both of which involve partitioning the original information into 2 dimensional components.

Each spectral band could be viewed individually. The data contained in that band is transformed into a 2d pixel array and displayed as a monochromatic photograph. (See Fig. 1.2a) The shade or brightness of each pixel indicates the intensity of reflected light at the given point. Each of the bands can then be displayed by a standard

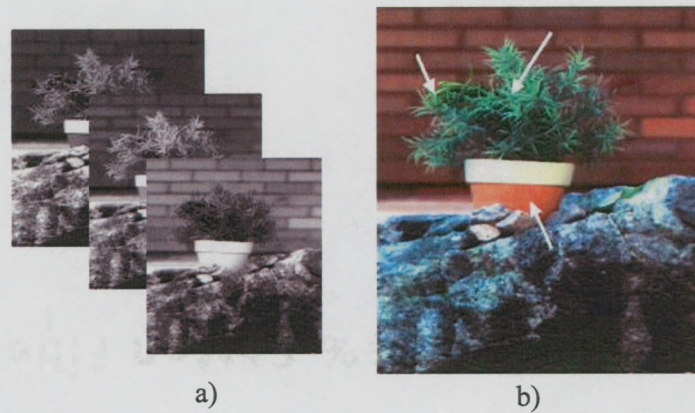


Figure 1.2: Visualization of a Hyperspectral Image: a) Collection of Monochromatic Images. b) Composite RGB Image. [9]

graphical toolkit, or saved in a standard lossless file format for processing in some mainstream graphical analysis software. The snapshots of individual bands can be visually compared, and contrasted providing useful clues for feature extraction, or other types of analysis.

Alternatively, 3 bands could be combined together to produce a composite 2d image in full RGB color. (See Fig. 1.2b) The photograph created in this way can be handled exactly like the gray-scale snapshot. If the bands chosen in the composition roughly correspond to red, green and blue light wavelengths, then the resulting pictured will be displayed in true color. Other combinations will produce completely artificial coloring. It has to be noted that artificially colored images may inadvertently produce odd visual artifacts because the choice of colors is completely arbitrary. For example swapping blue and green light frequencies may make a photo of a forest look like a lake.

In both cases, visualizing hyperspectral data they may convey some additional information. For example, vegetation may often look brighter in the near infra-red wavelengths, making it easy to distinguish it from other green objects in the image. This technique is often used to highlight certain features in satellite photography or

pictures of deep space objects.

Hyperspectral images with few or few dozen bands are usually referred to as Multispectral. These images are usually taken with smaller, less complicated sensors on the ground. Hyperspectral images on the other hand are usually taken by complex and sensitive satellite systems [8]. However there is not clear cut distinction between the two and they can be often used interchangeably. The accuracy and amount of extractable information grows proportionally to the number of spectral bands and thus the most desirable data sets are the hyperspectral images which are characterized both by high resolution, and large number of bands.

1.2.2 I/O and Encoding

Because of their nature, multispectral and hyperspectral images are often different than standard digital photography with respect to the way they are stored and encoded. Most digital images are stored in files, written according to a rigid standardized format such as JPG, PNG, GIFF, TIFF and etc.. These formats only describe how to encode a 2 dimensional array of pixels, representing a single band, and thus they do not apply to multi dimensional data used in remote sensing.

There are no standardized file formats defined for multi and hyper spectral data. Different spectrographic tools can produce very different outputs. In most cases hyperspectral files are simple data dumps, written according to one of the three basic methods. Multi-band information can be encoded in following ways: Band Interleaved by Line (BIL), Band Interleaved by Pixel (BIP), and Band Sequential (BSQ) [3].

In imaging a line (also known as scanline) can be defined as a vector of pixels, whose length is equal to the width of the image. It should not be confused with hyperspectral pixel vector. The values of a scanline are spatial neighboring pixels,

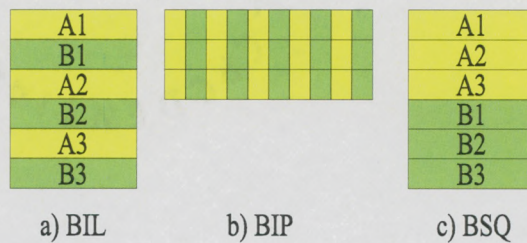


Figure 1.3: Different Encoding Methods: a) BIL - Band Interleaved by Line. b) BIP - Band Interleaved by Pixel. c) BSQ - Band Sequential.

while values of the pixel vector are spectral identities of each other. BIL encoding interleaves scanlines by band. Each actual line of hyperspectral the image can be therefore represented as a matrix in which each row is a scanline of one band, and each column is a pixel vector. A monochromatic image based on a single band can be generated by reading every n^{th} line (where n is the number of the band). See Figure 1.3a [3].

BIP encoding takes a different approach and interleaves bands by pixel. In other words, each line of the image is composed of consecutive pixel vectors. All the information about a given pixel is contained in it's respective scanline. A monochromatic view can be obtained by reading every n^{th} pixel of each line. See Figure 1.3b [3].

Finally, in BSQ each band exists as a separate gray scale image. These images are concatenated to each other, and written sequentially. See Figure 1.3c. All three methods are widely used, and neither one is more popular than the others [3].

The way in which bands are encoded in file is not the only issue when dealing with hyperspectral images. Pixel representation is another one. There is not standardized method for representing pixel values in a file. Therefore they can be encoded as bytes, integers (both signed and unsigned, 32 or 64 bit long), or floating point values (both 32 and 64 bit). The values of pixels can furthermore be in big endian or little endian

encoding. Altogether there are around 42 different combinations of band format, pixel representation, and endian encoding.

Handling some of the data types is more difficult than others, specifically because Java lacks support for unsigned integers and thus they have to be emulated using signed variables of higher order data type, and thus potentially increase the complexity of I/O modules.

1.3 Feature Extraction

Feature Extraction as applied in Hyperspectral Imaging field, roughly consists of two separate but related problems: dimensionality reduction, and unmixing.

1.3.1 Dimensionality Reduction

In most cases, the aim of feature extraction is to reduce the dimensionality of the processed sample without a significant data loss. In other words, the processing steps should transform an image with m bands into a smaller image with n bands where $m > n$ (See 1.4). The result should be a representative sample of the original, and contain all of its variance. In other words, reduction cannot be simply accomplished by arbitrarily dropping bands, as this would lead to significant information loss.

A desirable result of dimensionality reduction is an increase in separation between distinct classes (or types sources of reflected light) within the image [10]. In an ideal situation one would achieve ideal separation of classes by band, so that each source material could be viewed independently.

However, dimensionality reduction is only a part of the issue.

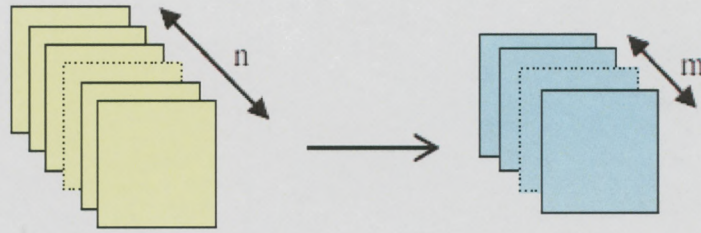


Figure 1.4: Reduction of Data Dimensionality [10]

1.3.2 Linear Mixing Model (LMM)

Hyperspectral images are very often satellite or areal photos. As such they suffer from spectral mixing. Each pixel of a hyperspectral image can contain more than one object or source material. The intensity of a given pixel then represents a combination of intensities of light reflected by all the materials which are contained in that pixel. Of course, some pixels can be for the most part classified as pure if it's evident that they contain only a single structural component. For example a pixel located in the middle of a calm lake can probably be quite accurately classified as "lake water".

Mixed pixels can usually be found on the edges of the pure areas or in areas with high density of non-identical objects. For example, a snapshot of the beach will usually contain a strip of pixels which contain both sand and water. In this example it is very easy to guesstimate potential pure and mixed parts of the image. However this is not always the case. A reversal of this scenario could be a picture of a dense forest. The foliage of different trees may produce different spectral signatures. Since different species of trees can grow very close to each other, one might expect the photo to be composed mostly of mixed pixels. In fact, it might be quite difficult to find any pure areas.

The feature extraction process has to be able to take the mixed pixels into account when separating and reducing the data set. This issue is commonly known as the unmixing problem. Good feature extraction algorithms for hyperspectral data must

be able to deal with unmixing.

Main assumption behind LMM is that each pixel is in fact composed of d primitive components (or endmembers) which all contribute some fractional amount to the end intensity of that pixel. Each endmember has therefore a corresponding abundance value, which describes the magnitude of this contribution. The actual intensity can be obtained by taking a dot product of abundance and endmember vectors. In this model a pure pixel would be one whose abundance vector contains only one nonzero value. Pixels close to pure would be those which either have few nonzero values, or a single dominant value.

The unmixing problem can be formally described in terms of the Linear Mixing Model (LMM).

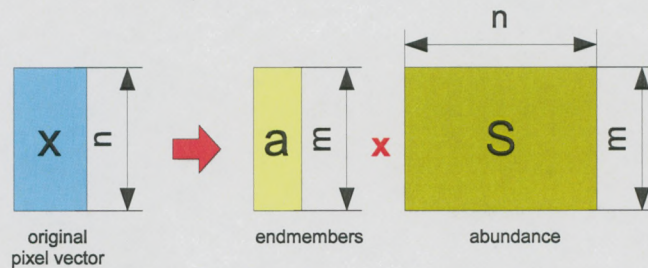


Figure 1.5: Linear Mixing Model

A hyperspectral n dimensional pixel vector x can be described as [11]:

$$x = \sum_{i=1}^m a_i s_i + w = Sa + w \quad (1.1)$$

where S is the $n \times m$ matrix of endmember spectra, a is an m dimensional vector of the fractional abundances and w is some additive noise (See Fig 1.5). All the elements of a are presumed to be positive, and their sum should be equal to one [11]:

$$a_i \geq 0, i = 1, \dots, m \quad (1.2)$$

$$\sum_{i=1}^m a_i = 1 \quad (1.3)$$

Feature extraction by unmixing in terms of LMM can be defined as finding end-members and their abundances, for a given multidimensional vector \mathbf{x} .

There are two general approaches to feature extraction: supervised and unsupervised.

1.3.3 Supervised Feature Extraction

Supervised extraction assumes that composition of processed image is at least partially known, and that certain features can be identified prior to processing. This can be easy in the beach example, where one can clearly identify a sample area of water, and of sand. This sort of visual classification can be conducted using specialist tools such as the MultiSpec software developed at Purdue University [12,13], or the ENVI suite produced by ITT [14].

If visual classification is impossible, or impractical, it is still possible to use supervised feature extraction, provided that some adequate training data is available. It can either be obtained from previous experiments, or from archives cataloging spectral signatures for various materials. For example, when processing a satellite photo of a forest one could use information about spectral properties of different types of tree foliage as inputs.

The reduction is accomplished by comparing the known data to the processed sample. Separation can be obtained by measuring the differences between spectral

signatures of each pixel and the set ones obtained from the inputs and appropriately clustering the pixels according to the results.

Supervised feature extraction is limited because it heavily relies on preexisting set of samples that can be used as training data. Results of the reduction are only as good as the starting dataset allows them to be [15].

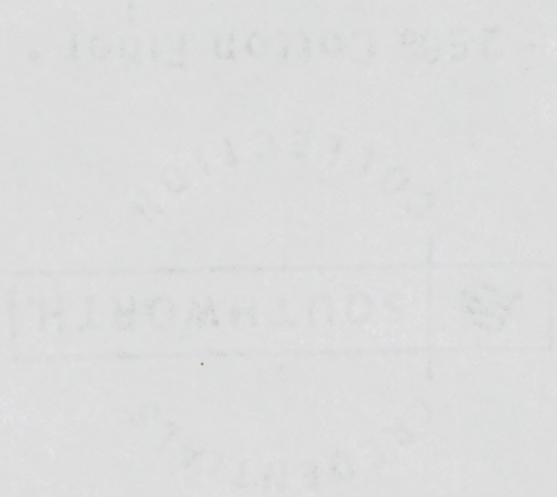
1.3.4 Unsupervised Feature Extraction

Unsupervised extraction is performed when no prior data is available or manual training is not possible. In many ways, the unsupervised approach is more attractive both to researchers, and for commercial applications because it allows for creating fully automatic hyperspectral data processing systems. It also eliminates the possibility of human error, skewing the results. Because of the absence of training samples, the images must be analyzed using purely statistical methodology. In other words, the unsupervised methodology concerns itself only with the data at hand, without any inputs from the outside [15].

Consequently, the main focus is not comparison, but rather reduction of data redundancy. The data is processed by applying various mathematical transforms to the original set which reduce its dimensionality and increase data separation.

There are several interesting feature extraction algorithms but in the following chapters we will mainly concentrate on the Nonnegative Matrix Factorization methods. We will devote special attention to the original algorithm as defined by Lee and Seung [1] and its proposed implementation in [16] as well as the highly optimized projected gradient method described by Chih-Jen Lin in [2]. We will briefly discuss other related implementations of NMF in section 2.1.5 and also other competing feature extraction methods such as Principle Component Analysis (PCA) and Independent Component

Analysis (ICA) in section 2.2.



Chapter 2

Nonnegative Matrix Factorization

Methods

2.1 Nonnegative Matrix Factorization (NMF)

2.1.1 NMF: General Problem Description

Nonnegative Matrix Factorization is a good example of an unsupervised feature extraction method. Lee and Seung claim that what distinguishes NMF from other extraction methods such as Principal Component Analysis, is its parts-based approach. While other methods take a more holistic approach, NMF performs well in discovering features of objects by learning about their parts. They suggest using it for tasks such as facial recognition, or discovering semantic features of text [1]. Chu and Plemmons however show that it can be adapted to other domains as well. They specifically recommend it for image and spectral data processing, text mining and air emission quality problems [17]. Paucca, Piper and Plemmons develop an NMF based algorithm especially geared for hyperspectral data in [16]. This algorithm will be

described in greater detail later. Before that, it is best to define the NMF problem, and talk about it more thoroughly.

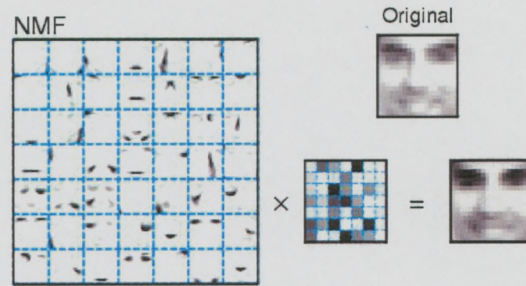


Figure 2.1: Visual Example Non Negative Matrix Factorization [8]

A good visual representation of the NMF algorithm at work can be seen in Figure 2.1. There, it is clearly visible how the original image of a face, is decomposed into a large array of partial facial features such as eyes, noses, lips and etc.. and a mapping of how these parts contribute to the original image. These arrays correspond to end-members (W) and abundances (H) from the Linear Mixing model. This compatibility with LLM is one of the major advantages of NMF over alternative feature extraction methods. Other notable advantages are it's relative simplicity and the low complexity of basic calculations it requires.

As mentioned before, NMF is designed to work with nonnegative inputs so all the processed data must be greater or equal to zero. As long as that requirement is fulfilled NMF will always produce nonnegative results. It is highly desirable because it guarantees that equations 1.2 and 1.3 will hold. This is not always true for other methods. Fulfilling the requirement of always using nonnegative inputs is not difficult, since most imaging data is by nature either zero or positive.

NMF problem can be formally defined as follows [1, 16, 17]:

Given a nonnegative $m \times n$ matrix Y and a positive integer k such that:

$$k \leq \min\{m, n\} \quad (2.1)$$

find two nonnegative matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ such that their product approximates Y :

$$Y \approx WH \quad (2.2)$$

while minimizing the functional:

$$f(W, H) := \frac{1}{2} \|Y - WH\|_F^2 \quad (2.3)$$

where $\|\cdot\|_F$ denotes a Frobenius (or Euclidean) norm. W is the matrix of endmembers, and H is the matrix of abundances (See Fig. 2.2).

The product of W and H will be of rank of at most k and almost never actually be equal to Y . The choice of the integer k is often a very critical choice, but it is usually problem dependent. The equation 2.3 can be modified to better reflect the given situation. For example, penalty terms could be added to $f(W, H)$ in order to enforce sparsity or to enhance smoothness in the solution matrices [16, 17].

In addition, because $WH = (WD)(D^{-1}H)$ for any invertible matrix $D \in \mathbb{R}^{k \times k}$ sometimes it might be desirable to normalize the columns of W [16, 17].

It has been experimentally shown that NMF performs well as a feature extraction method and provides an elegant solution to the unmixing problem. Other approaches such as PCA and ICA are focused on strong restrictions on the separability of the resulting bands, and do not have a natural interpretation for the nature of hyperspectral data. Compared to them, NMF is much less restrictive simply assuming

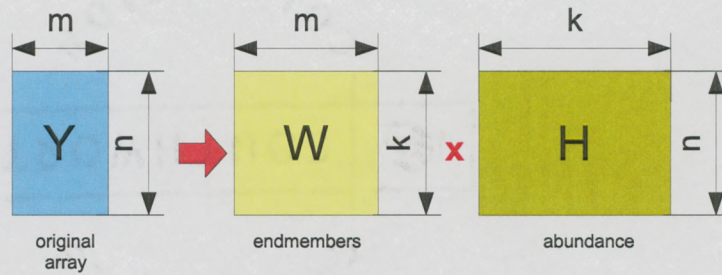


Figure 2.2: Nonnegative Matrix Factorization

that features must be separable, and positively defined [10].

2.1.2 Applying NMF to Hyperspectral Data

A hyperspectral image can usually be conceptualized as a 3D cube created by stacking individual spectral band images on top of each other. It could be relatively easily represented as a cubical data model represented as a 3D matrix (a matrix of 2D matrices). Unfortunately such a representation would be overly complex to work with and technically unusable. Most feature extraction algorithms (including NMF) are designed to work with 2 dimensional data structures which can be represented with matrices. Thus the data must be transformed into a more conventional format before any processing is done.

One way to avoid unnecessary complexity is treating each band as a linear vector (or bandvector) rather than a matrix. A bandvector can be created by concatenating all the scanlines of a given band. The whole image then can be treated as a two dimensional $m \times n$ matrix Y where m is the total number of pixels in a single band image, and n is the total number of bands (See Fig. 2.3).

In Y the spatial relationship between pixels is lost. However, if the original row length is retained somewhere, then Y can be easily converted back to the 3D form by simply cutting up the rows, and stacking them up.

2.1.3 Standard Iterative NMF Algorithm

Applying an iterative NMF algorithm to a general unmixing/feature extraction problem was first proposed by Lee and Seung in [1]. Later it was refined to be used on Hyperspectral data by Pauca, Piper and Plemmons in [17]. The algorithm presented below is a refinement largely based on these ideas. The algorithm adapted from [17] was experimentally tested and shown to produce good feature separation in [10] and [9]. The tests were conducted on a sequential iterative algorithm much like the one presented below.

The algorithm can be characterized as diagonally scaled gradient descent method [16]. It is based on minimizing the distance between Y and iterations of WH by repeatedly updating the values of W and H [9,10,18]:

$$W = W - \frac{\partial f(W, H)}{\partial W} \quad (2.4)$$

$$H = H - \frac{\partial f(W, H)}{\partial H} \quad (2.5)$$

The function $f(W, H)$ is of course the same one as defined in equation 2.3.

An elegant solution that ensures positivity is described in [18] for which the update

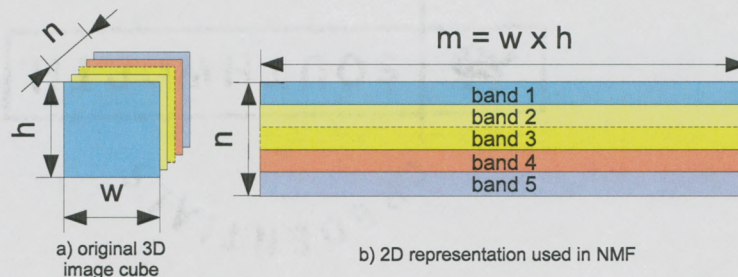


Figure 2.3: Hyperspectral Data and NMF

steps are:

$$W = W \frac{(YH^T)}{(WHH^T) + \epsilon} \quad (2.6)$$

$$H = H \frac{(W^TY)}{(W^tHWH) + \epsilon} \quad (2.7)$$

These steps are relatively straightforward and can be easily implemented in code. Thus the NMF algorithm can be formally defined in Figure 2.4.

The NMF Algorithm

1. Given

$$Y \in \mathbb{R}^{m \times n} \geq 0, k > 0 \quad \text{and} \quad k \ll \min(m, n) \quad (2.8)$$

randomly initialize matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ with nonnegative values

2. Scale the columns of W to sum up to one (to satisfy eq. 1.3)

3. Create temporary variables \bar{W} and \bar{H} . Set their contents to be equal to H and W respectively.

4. Repeatedly apply the following steps until convergence criteria are met:

(a) Update \bar{W} and \bar{H} by using:

$$\bar{H}_{cj} \leftarrow H_{cj} \frac{(W^TY)_{cj}}{(W^tWH)_{cj} + \epsilon} \quad (1 \leq c \leq k) \quad (1 \leq j \leq n) \quad (2.9)$$

$$\bar{W}_{ic} \leftarrow W_{ic} \frac{(YH^T)_{ic}}{(WHH^T)_{ic} + \epsilon} \quad (1 \leq i \leq m) \quad (1 \leq c \leq k) \quad (2.10)$$

(b) Set $W = \bar{W}$ and $H = \bar{H}$

(c) Scale the columns of W to sum up to one

Figure 2.4: NMF Algorithm

The algorithm requires that operations 2.9 and 2.10 and the scaling step are repeated until the values of W and H converge. The exact criteria for convergence may vary and are discussed in Sect. 2.1.6. For now we can assume that there exists an

explicit condition that will cause the execution to stop, and result in optimal source separation.

Please note that temporary variables are used in order to store the data during the calculation. This way, values in each step are calculated using a 'clean' matrices H or W as a source. While this takes up memory, and creates overhead due to matrix copying it ensures data integrity. In other words each new set of matrices is computed based using data solely from the previous iteration.

The ϵ value is a very small positive quantity. As suggested in [16] an optimal starting value is $\epsilon \leq 10^{-9}$. The size of ϵ can be used to tweak the accuracy of the algorithm. It is possible for this algorithm to converge on a local optima that is not the best approximation of Y. Increasing, or decreasing the size of ϵ can help to find the optimal convergence [1, 16].

The computational complexity of this algorithm has been shown to be $O(kmn)$ per iteration [16]. Since the number of iterative steps is usually unknown, or dependent on convergence criteria, the total time of execution for very large images can be very long.

2.1.4 Projected Gradient NMF (PG-NMF)

Lee and Seung work in [18] proves that the function f (equation 2.3) is non-increasing after every update. From that property it follows that NMF could be solved by alternatively fixing one matrix and updating another. Thus the problem can be redefined as [19]:

$$\text{find } W^{k+1} \text{ such that } f(W^{k+1}, H^k) \leq f(W^k, H^k) \quad (2.11)$$

and

$$\text{find } H^{k+1} \text{ such that } f(W^{k+1}, H^{k+1}) \leq f(W^{k+1}, H^k) \quad (2.12)$$

This approach can be categorized as a block coordinate descent method in bound constrained optimization [20] where one block of variables is minimized under corresponding constraints and the remaining blocks are fixed. Of course in our example we have a very simple case with only two block variables [2].

Since bound constrained optimization problems are very efficiently solved by projected gradient approach Chih-Jen Lin in [2] proposes to apply it to the NMF problem. He starts off with a generalized problem for bound constrained optimization:

$$\text{Find } \min f(x) \text{ where } x \in \mathbb{R}^n \text{ and } l_i \leq x_i \leq u_i \text{ and } i = 1, \dots, n$$

The update rule for x can be defined as follows:

Given $0 < \alpha < 1$, $0 < \beta < 1$ and randomly initialized x compute:

$$x_{k+1} = P[x^k - \alpha_k \nabla f(x^k)] \quad (2.13)$$

where:

$$P[x_i] = \begin{cases} x_i & \text{if } l_i < x_i < u_i \\ u_i & \text{if } x_i \geq u_i \\ l_i & \text{if } x_i \leq l_i \end{cases}$$

and $\alpha_k = \beta^{t_k}$ where t_k is some non-negative integer t for which:

$$f(x^{k+1}) - f(x^k) \leq \alpha \|\nabla f(x^k)^T (x^{k+1} - x^k)\| \quad (2.14)$$

The α should be updated using following criteria:

1. if α_k satisfies 2.14 then repeatedly increase $\alpha_k \leftarrow \alpha_k/\beta$ as long as it still satisfies 2.14.
2. else repeatedly decrease $\alpha_k \leftarrow \alpha_k * \beta$ until it satisfies 2.14.

This general framework can be applied to NMF problem. Chih-Jen Lin defines the H updates as follows [2]:

$$H_{k+1} = P[H_k - \alpha \nabla f(H_k)] \quad \text{where} \quad \nabla f(H) = W^T(WH - Y) \quad (2.15)$$

In a more concise form, updates for both matrices can be defined as:

$$H_{k+1} = P[H_k - \alpha W_k^T(W_k H_k - Y)] \quad (2.16)$$

$$W_{k+1} = P[W_k^T - \alpha H_{k+1}(H_{k+1}^T W_k^T - Y^T)] \quad (2.17)$$

The α update rule 2.14 can be rewritten for H:

$$(1-\sigma)\langle W_k^T(W_k H_k - Y), H_{k+1} - H_k \rangle + \frac{1}{2}\langle H_{k+1} - H_k, (W_k^T W_k)(H_{k+1} - H_k) \rangle \leq 0 \quad (2.18)$$

and for W:

$$(1-\sigma)\langle H_{k+1}(H_{k+1}^T W_k^T - Y^T), W_{k+1}^T - W_k^T \rangle + \frac{1}{2}\langle W_{k+1} - W_k, (H_{k+1} H_{k+1}^T)(W_{k+1}^T - W_k^T) \rangle \leq 0 \quad (2.19)$$

In the above, the \langle, \rangle denotes the sum of component wise products of two matrices. The recommended value of σ is 0.01 The Projected Gradient Nonnegative Matrix Factorization is defined in Figure 2.5.

PG-NMF Algorithm

1. Given

$$Y \in \mathbb{R}^{m \times n} \geq 0, k > 0, k \ll \min(m, n), \alpha = 1, \beta = 0.1, \sigma = 0.01 \quad (2.20)$$

randomly initialize matrices $W \in \mathbb{R}^{m \times k}$ and $H \in \mathbb{R}^{k \times n}$ with nonnegative values.

2. Find H_{k+1} using Equation 2.16

3. Evaluate Equation 2.16 and:

(a) if Equation 2.18 is satisfied then:

- i. if at the last iteration Equation 2.18 was not satisfied set $H_{k+1} \leftarrow \bar{H}_{k+1}$ and goto 4
- ii. else save the value of H_{k+1} in a temporary buffer \bar{H}_{k+1}
- iii. save the outcome of Equation 2.18
- iv. update $\alpha \leftarrow \alpha/\beta$
- v. go back to 2.

(b) if Equation 2.18 is not satisfied then:

- i. if at the last iteration Equation 2.18 was satisfied set $H_{k+1} \leftarrow \bar{H}_{k+1}$ and goto 4
- ii. else save the value of H_{k+1} in a temporary buffer \bar{H}_{k+1}
- iii. save the outcome of Equation 2.18
- iv. update $\alpha \leftarrow \alpha * \beta$
- v. go back to 2.

4. Find W_{k+1} using Equation 2.17

5. Evaluate Equation 2.16 and perform steps analogous to 3a and 3b for W .

6. Set $H = H_{k+1}$ and $W = W_{k+1}$

7. Go back to 2 until convergence criteria are met.

Figure 2.5: PG-NMF Algorithm

Please note that in the above algorithm initially W is fixed, while H is repeatedly

updated to find the best value. Next, H is kept fixed while W is updated. Such approach in some cases may be more desirable than standard NMF because the total number of iterations required to reach convergence might be smaller. On the other hand, compared to the other algorithm PG-NMF uses a considerably more complex (and thus time consuming) procedure to find H_{k+1} and W_{k+1} .

Lin's algorithm is one of the most optimized NMF implementations based on the bound constrained optimization model. He was able to drastically reduce the cost of inner iterations by taking advantage of the unique formulation of this problem. Experiments conducted by Ingram, seem to suggest that this approach can be decidedly faster than Lee and Seung's multiplicative approach [21]

The initial α , β and σ values were chosen based on the recommendations outlined in [2]. Since the focus of this project is investigating parallelization of NMF algorithms we will simply assume that these values are optimal in all cases.

2.1.5 Other NMF Approaches

There are several other notable approaches to NMF problem which were not our main focus. This paper mainly concentrates on standard NMF as defined in [16] and the PG-NMF variant developed by [2]. However, some approaches are especially noteworthy.

2.1.5.1 Stochastic NMF (S-NMF)

The original algorithm in [16] did not implicitly recommend using temporary matrices for W and H during the updates. They were included in our implementation of the algorithm because of concerns about data integrity and concurrency conflicts. However conducting updates on live data is an attractive alternative. In theory,

stochastic approach could potentially speed up the time required for convergence as updated values would become available immediately rather than after each iteration hopefully creating a much smoother curve of function 2.3.

The algorithm for S-NMF would be identical to the one defined in 2.1.3 but instead of using temporary variables the updates would be applied directly to H and W.

2.1.5.2 Multilayer NMF

Performance of multiplicative and projected gradient NMF algorithms such as the ones presented in Section 2.1.3 and 2.1.4 can often be poor. This is especially true when the unknown nonnegative components are badly scaled, insufficiently sparse or number of observations is equal or only slightly greater than a number of latent (hidden) components [22]. This is one of the reasons why we are developing parallel implementation of these algorithms in this paper.

However Cichocki, and Zdunek in [22] propose a solution which would increase the performance. Their idea is relatively simple. Initially a decomposition step is conducted using any available algorithm, be it standard NMF, PG-NMF or some other implementation yielding:

$$Y = H_1 W_1 \tag{2.21}$$

The next step would then be conducted on the obtained set of results rather than on the original matrix Y:

$$H_1 = H_2 W_2 \tag{2.22}$$

The final model for the composition would be as follows:

$$Y = H_1 H_2 H_3 \dots H_L W_L \quad (2.23)$$

The authors claim that this approach can improve performance and accuracy of most NMF algorithms, and mention that they concluded extensive simulations to prove it. Unfortunately they did not include any experimental data or results in [22] so it is difficult to judge the effectiveness of their design.

2.1.5.3 PG-NMF with Initial Newton Step

In [21] Ingram proposes to extend the Chi-Jen Lin's algorithm by adding some extra steps that can further optimize the performance. He notes that the Hessian matrices of W and H are $W^T W$ and $H H^T$ respectively. These matrices are both $k \times k$ and positive definite. Therefore it makes sense to add a Newton direction into the Lin's line search algorithm. It can be accomplished by inverting the the Hessian matrix, multiplying the gradient and performing a projected line search.

Unfortunately the Newton step can only be used on free (non-zero) variables. If any value of W or H becomes zero, its corresponding column of the Hessian must be excluded from the computation. Excluding columns from these matrices however would not make sense in PG-NMF framework. Therefore Ingram proposes to test values of W and H at each iteration, and only compute Newton direction if they are non-zero.

He argues that the benefits of this added step will offset the overhead involved with testing for non-zero values. The cost of such a test is $O(mk)$ and $O(nk)$ respectively and thus it can be performed relatively quickly. In addition most algorithms initialize W and H to random non-zero noise, so at least the initial few iterations could take full benefit from the Newton direction computation. When zero values are found, we

should simply fall back to Lin's standard PG-NMF implementation [21].

Ingram's experiments suggest that while this method can improve the quality of the results, and the overall performance it does not always do so. In some cases the algorithm was shown to yield worse results than PG-NMF. In addition the costs of computing a Newton step on data sets with a very large k can be very expensive. Ingram suggests that his approach is best suited for real-time processing problems in which reduced quadratic model is inadequate (such as robotics, signal processing and etc..) [21].

2.1.6 Convergence Criteria

Choosing appropriate convergence criteria is very important. If they turn out to be too lax, then processing may be stopped prematurely, producing incomplete or sub-par separation. On the other hand, if criteria are too strict, they may never be reached and the program may enter an infinite loop. Such a situation would happen if our criteria would unknowingly exclude the global optima.

Finding good convergence criteria is not trivial, and there are no algorithmic methods to find them. In most cases, the criteria are problem dependent, and may need to be tweaked based on experimental observation. Thus, we decided it would be best to leave the setting of convergence criteria up to the user. We have defined 4 different ways in which our program can converge, and the user can adjust the exit values in each category.

The best indicator of convergence is of course the value of $f(W,H)$ as expressed in equation 2.3. It is the measure of the quality of the NMF extraction. The smaller the value, the better the results. Unfortunately while $f(W,H)$ generally seems to approach zero in most cases, it is not always decreasing (see fig. 2.1.6).

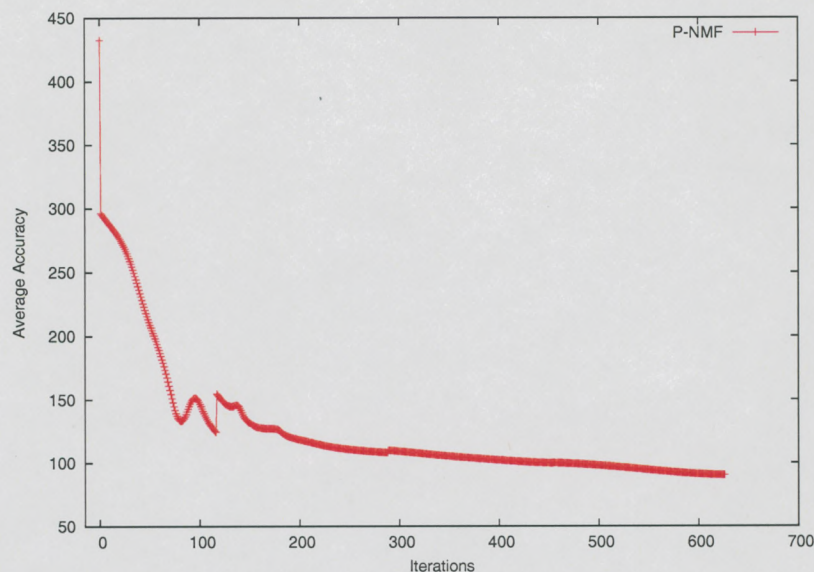


Figure 2.6: Sample Graph of $f(W,H)$

Please note that as the number of iterations increases, the value of f seems to be increasing and decreasing, creating many peaks and valleys on the graph. Frequently, f will reach some local minimum, and then start increasing, only to stop at a certain threshold and start decreasing again. Over time, the local minima seem to become smaller. Thus the more iterations we allow, the better are chances for finding the absolute minimum, of the function.

Based on these properties, we have developed 4 different criteria which can be used when testing for convergence:

1. **Fixed Convergence Target** - a fixed desired target value of $f(W,H)$ can be chosen. This target is the estimated value of the absolute minimum of the function. The execution will then continue until that target is reached, or until it is stopped for some other reason. Choosing a good target value is very crucial. Poor target can cause the program to either stop prematurely, or enter infinite loop.
2. **Change of $f(W,H)$** - because it is hard to pick a perfect convergence target, it

might be beneficial to track how the value of $f(W,H)$ changes at each iteration. While it is increasing or decreasing from one iteration to the other, it can be assumed that convergence has not been reached yet. Once it stops changing, it is safe to assume that an optima has been reached. It also is worth noting that the magnitude of change decreases as the function is minimized. By picking a good delta value it is possible to get good distribution in reasonable amount of time. This might be hard however, because the algorithm may reach a local optima and change very slightly over many iterations. If it is stopped then it will yield an incomplete separation. As evidenced by our experiments, a good starting target value is 10^{-5} which would yield good separation in the output matrices and yield reasonable number of iterations.

3. **Number of Iterations** - in case both criteria based on evaluation of $f(W,H)$ are poorly chosen, the user should also have a chance to set the maximum number of iterations that will be executed before the program terminates. This prevents unwanted infinite loops caused by poorly chosen convergence values.
4. **Time of Execution** - time of execution measured in hours or minutes may be an alternative to number of iterations.

It should be possible to find a good stopping conditions for most data sets using combinations of the four criteria described above.

2.2 Other Methods

Nonnegative Matrix Factorization is not the only feature extraction method. Analyzing the wide range of existing algorithms would be out of scope of this paper. We refer the reader to the quantitative analysis of various non-NMF extraction algorithms conducted by Plaza et all in [23]. In this section we will only describe

two most notable competing feature extraction algorithms: the Principle Component Analysis and Independent Component Analysis. We will also explain why we decided to choose NMF over these attractive alternatives.

2.2.1 Principal Component Analysis

One popular method that can be used in feature extraction is Principal Component Analysis. PCA is a linear transformation that transforms the data to a new set of coordinates in a way such that the greatest variance lies on the first few of them. These coordinates are called the Principal Components of the data set, and they are usually uncorrelated and will usually contain most of the variation present in all of the original data [24].

The PCA problem can be defined as finding a linear transform W for a multi-dimensional vector x such that the obtained components are uncorrelated:

$$Y = Wx \quad (2.24)$$

In equation 2.24 Y represents the Principal Component vector. The transform W is obtained as:

$$W = A_x^T x \quad (2.25)$$

where A is a matrix of normalized eigenvectors for the covariance matrix Σ_x . The eigenvalues of Σ_x correspond to the variances of the principal components. When these values are sorted in decreasing order, the values of Y also become sorted in decreasing order with respect to their variance [15].

PCA can be used to reduce dimensionality of the sample by selecting the compo-

nents with the highest variance, and ignoring the rest. The components of interest will usually be contained the first few variables, but depending on the implementation approach, they could also potentially appear in the few last ones. The quality of the reduced sample greatly depends on the correct choice of components. If too few are chosen, then the new sample will not be representative of the original set. If there are too many, then it is possible that there is still redundancy and noise in the sample. Unfortunately, selecting which Principal Components are significant and which can be ignored is not a trivial task.

There exist many different methods which aim to make this choice automatically. One can select the correct components by looking at cumulative percentage which they contribute to the total variation, or by looking at respective size of variance of each component. These methods however are mostly ad-hoc rules of thumb which generally work in practice but lack adequate formal basis [24].

Principal Components can also be found by more formally grounded hypothesis testing procedures. Ian T. Jolliffe describes some of these methods in his book on Principal Component Analysis, but he claims that usually they pose unrealistic distributional assumptions and tend to retain more components than it is necessary [24].

Third category of PC selection methods depend on statistical analysis and cross-validation. According to Jolliffe most existing algorithms in this category are relatively slow and very computationally intensive [24].

It is also noted that PCA can sometimes be less efficient when dealing with small classes within the image. Since small classes do not contribute to band variance in any significant way, and thus they could be only featured in low variance components, and thus get discarded during Principal Component selection. This loss of information can be very problematic, especially if one would want to use PCA for target detection,

when dealing with very small targets [10].

2.2.2 Independent Component Analysis

Another method which could be used for feature extraction is Independent Component Analysis (ICA). It stems from a more general problem called Blind Source Separation (BSS). In BSS one deals with a multivariate signal produced by several unknown sources such as people speaking over each other to a microphone. The aim is to recover the signal produced by each source without knowing the way they were mixed. In other words, given an n dimensional vector x containing all the observed values we want to find an m dimensional vector s in which each value corresponds to a single source, and a $n \times m$ mixing matrix A [15]:

$$x = As \tag{2.26}$$

Unfortunately, for a given x , there may exist an infinite number of pairs of s and A which satisfy 2.26. However, it has been shown, that an unique solution for 2.26 can be found if it is assumed that all the components of s have non-Gaussian distribution, are statistically independent (hence the name Independent Component Analysis) and that their probability density function $p(s)$ can be represented as [15]:

$$p(s) = \prod_{i=1}^m p(s_i) \tag{2.27}$$

The solution can be found either by minimizing the mutual information of its components as expressed by:

$$I(s_1, \dots, s_m) = E\left\{\log \frac{p(s)}{\prod_{i=1}^m p(s_i)}\right\} \quad (2.28)$$

or by maximizing their non-Gaussianity, while keeping them decorrelated [15].

ICA is well suited to work in hyperspectral imaging, because the BSS can be easily converted into an unmixing problem where each source is a class within the image.

In most scenarios PCA and ICA perform similarly, and could be used interchangeably. However when dealing with non Gaussian, ICA provides much stronger decorrelation. [10, 25] Some sources suggest that PCA and ICA can be easily combined for better effect. PCA could be used as a preprocessing step which would reduce dimensionality of the sample, which in turn could be processed using ICA to achieve good source separation [15, 25].

Similarly to PCA, choosing the correct Independent Components in ICA is problematic. Dimensionality reduction can be accomplished by choosing the most non-Gaussian ones with respect to their kurtosis, and discarding the rest [10].

2.2.3 Linear Mixing Model and PCA/ICA Feature Extraction

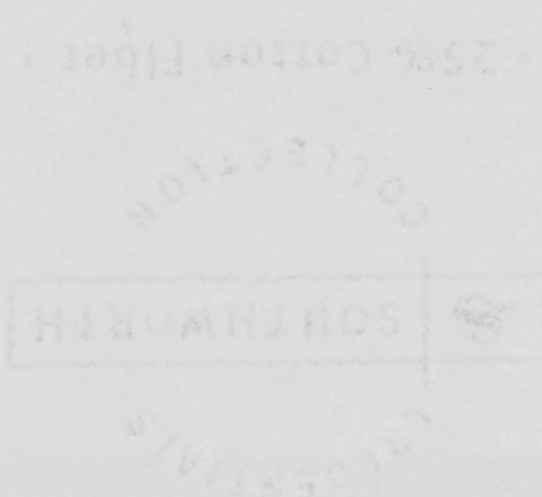
While both PCA and ICA can be used to perform feature extraction on graphical data, neither one is ideally suited to work with hyperspectral images which conform to the LMM. There are several noteworthy issues which must be addressed when attempting to use these methods.

Neither PCA nor ICA have an efficient and clearly defined algorithm to find the number of endmembers present in the original image. In both cases the methods used

to pick the most significant components either involve either an ad-hock approach based on educated guesswork, or highly inefficient post-processing [10].

In PCA and ICA the endmembers correspond to rows in the linear transform matrix, and are orthogonal. This condition often proves to be too strict when doing endmember extraction on materials which may be very similar to each other [10]. This is often the case in hyperspectra photographs where one might find large concentrations of closely related sources such as different types of vegetation in a forest. The NMF's parts based approach handles these special cases much better.

Finally, neither PCA nor ICA can guarantee that equations 1.2 and 1.3 will hold [10]. Because of these issues, it is recommended to use different methods for LMM data.



Chapter 3

Parallel Approaches and NMF Algorithms

3.1 Different Parallel Approaches

3.1.1 The Need for Parallel Implementation of NMF

A major shortcoming of the NMF algorithm is that its execution time grows proportionally to the size of processed sample. According to [16] the standard NMF algorithm conducts $O(kmn)$ operations per iteration. Experimental results from [9] suggest that the number of iterations necessary for convergence can be very large, counting in the hundreds. The time of execution was shown to vary depending on the size of initial sample. Smaller multispectral images are usually processed quickly, but high resolution hyperspectral photos would take hours to be completely factorized.

The performance could be greatly improved if the two matrices W and H would be processed simultaneously. The Lee and Seung algorithm, can be easily very parallelized without introducing major changes. The operation 2.9 and 2.10 could simply

be placed on two different CPU's via threading or another parallelization technique. Both processes would work independently, and would only need to exchange the copy of it's respective matrix at each iteration. Since both calculations are about equally CPU intensive the idle time for each CPU would be minimal, and the execution time would be roughly cut in half. The example with 2 processing nodes is of course very simplistic. It is worth while to investigate the full extent of benefits of parallel processing for NMF.

3.1.2 Overview of Parallel Architectures

Parallel hardware architecture is another important factor which will impact the implementation. There are at least two distinct types of parallel architectures that need to be taken into account: Shared Memory and Distributed Memory. Each of these requires a slightly different approach.

Shared Memory systems, also known as Symmetric Multiprocessor (SMP) are characterized by the fact that all parallel CPUs' share common memory. Each processor has access to all memory locations using standard load operations. The cache is not shared, but specialized hardware is present to keep them synchronized and prevent conflicts [26]. The CPU scheduling is usually transparent to the user, and done at the operating system level. When using Java, the programmer is even more removed from the hardware implementation. When a Java application generates concurrent threads most modern JVM implementation should try to leverage OS capabilities to schedule them on different processors if possible. Thus, a considerable speedup can be expected.

SMP's are not very scalable. Usually, the number available processing units in this type of architecture is low, because the cost of cache synchronization, and memory collision avoidance grows exponentially with the number of processors. The most

powerful computers of this type have up to 32 (or in rare cases up to 128) nodes [26]. Most of the commercially available systems however are limited to 2 or 4 CPU's. Conversely, the communication overhead between processing nodes is probably the smallest when compared to other architectures. Working with SMP architecture will be the primary focus of this thesis.

Distributed Memory architecture is usually a massive array of independent processing nodes, each with it's own memory. Access to remote memory is very complex in this type of system. Most implementations rely on some sort of message passing programming models (such as Message Passing Interface or MPI) in which CPU's can exchange data among each other [26]. The communication is usually done through a dedicated, fast network connections, or via proprietary high speed communication interface. Standard local area connection, or even internet can be also used as communication media, but at the cost of performance. On average Distributed Memory systems will have a high communication overhead, but are almost infinitely scalable.

There also exist hybrid systems which combine the two approaches described above. Most notable are Distributed Shared Memory (DSM) systems and SMP Clusters. DSM architecture allows direct access to remote memory via some dedicated interface. As one can expect most implementation of DSM and proprietary and expensive. SMP Clusters on the other hand try to connect multiple SMP systems together to create a large distributed memory computer. They can be treated almost exactly like a standard distributed system, only with 2 CPU nodes. Many message passing interfaces are designed to seamlessly support this type of architecture [26].

Java does not provide a native message passing interface, but there are successful implementations that could be employed in the research project. One such implementation is JPVM developed at Virginia University. It provides functionality and performance equivalent to another message passing interface called PVM that has

existing implementations in C and Fortran [27]. It could be adopted to implement NMF on a Beowulf cluster or another distributed architecture.

The least sophisticated distributed system would be an ad-hock cluster of general purpose workstations, connected via non-dedicated, multi-purpose TCP/IP network. Each workstation would run it's own OS and a copy of the algorithm, and communication would be done either via Java Remote Method Invocation (RMI) calls, sockets communication or both. The communication overhead on that type of system is expected to be very large, as the processing nodes do not have to be in close physical proximity to each other, or even n the same subnet. Unlike dedicated clusters and SMP's however, this architecture is inexpensive, infinitely scalable, and does not require much hardware.

3.2 Data Distribution

An important issue for parallelization is the distribution of the processed information. There are two basic methods that can be used to distribute the workload among many nodes: spatial-domain partitioning, and spectral-domain partitioning [6, 28, 29].

The former method, as the name suggests divides the hyperspectral cube by cutting the actual image into n sub-images. Each of these subdivisions is an actual hyperspectral image in it's own right. Thus, each processing node is required to work only with a small image which is only a fraction of the original. The borders of sub-images will need to be swapped among the nodes processing neighboring segments if needed. This provides very natural framework for most processing algorithms, because all the the pixel vectors remain intact [6].

Spectral-domain partitioning on the other hand separates the original cube into individual band images. Each node gets to work with it's own gray scale image

representing a single band. Since each sub-image contains all the spatial data of the original, there are no border cases to be communicated. However, if the processing algorithm needs to access whole pixel vectors, this method may also require extensive communication [6].

NMF algorithms are best parallelized using spectral domain partitioning, because they perform most of their calculations on bandvectors, rather than pixel vectors. Plaza suggests that this approach is the most appropriate one for most of the morphological methodologies similar to NMF [28]. Spatial domain partitioning did not seem appropriate in any of the tested cases. It would require too much communication overhead and synchronization, as compared to the relatively communication-light spectral domain partitioning method.

3.2.1 Evaluation Criteria: Speedup

The performance improvement of a parallel implementation over sequential implementation can be measured using the concept of speedup factor. It can be calculated using the following ratio:

$$S_p = \frac{T_s}{T_p} \quad (3.1)$$

where T_s is the time of execution of the sequential algorithm, and T_p is the time of execution of the same algorithm using p processors. An ideal parallel implementation would of course yield speedup equal to p (also known as linear speedup). In most cases however speedup factor will be somewhere within the range of $1 < S_p < p$ [30].

Since the algorithm is divided into more than two parts, each processor will only work on a limited data set. It is safe to assume that there will be much more communication between the nodes required than in the 2 CPU model described above.

At each iteration, the H and W matrices will have to be synchronized. Thus, we will never achieve a linear speedup. However, a significant performance improvement can be expected.

3.3 Previous Efforts

Distributed hyperspectral image feature extraction is an active research field. However, most of the noteworthy work in that area is done with cluster architectures. Furthermore, there is very little interest in developing parallel implementations of NMF algorithms. In this paper we took the road less traveled and chose to investigate parallel NMF but it is worth mentioning the related research done in this field.

Achalkul and Taylor implemented and tested a parallel version of the PCA algorithm (see Section 2.2.1 in [31]). Their algorithm is designed for a hybrid architecture that involves networked, SMP machines working as a cluster. Unlike our own approach, Achalkul and Taylor decided to utilize spatial data distribution. In other words, the original image cube is divided into several sub cubes which can be operated on independently. The allocation of sub cubes is managed by a central managing (master) node that partitions the image, initiates worker threads, and collects partial results.

Plazza et al also opted for spatial data distribution in their parallel morphological algorithms in [29] and [6]. Their work also revolved around a cluster architecture. The communication overhead in such a system is significant, and therefore using spatial scheme allowed them to minimize the amount of data that needs to be sent from one node to another. In our approach we had no communication overhead and thus spectral partitioning became a viable option.

3.4 Parallel NMF Algorithms

Following sections will outline the implementation of the parallel algorithms for standard NMF and Projected Gradient NMF, as well as proposed Stochastic NMF approach. First, we will cover the implementation details, and challenges involved with creating a parallel implementation of complex algorithms. Then each algorithm will be discussed in details with respect to data distribution, synchronization and etc..

3.4.1 Implementation Details

The platform for implementation of our parallel algorithms was Sun's Java 1.4 [32]. This version was chosen over the newer Java 5 platform because it has a potentially larger install base than the newer version. As mentioned in the introduction one of the goals of this project, was to create a multi-platform implementation able to run on a wide array of systems without need for additional setup or installation. Thus we chose Java 1.4.

SMP was chosen as the target architecture in order to leverage Java's powerful Threading implementation. Java message passing algorithms for distributed systems exist [27], but are neither standard nor widely used. On the other hand, threading is a native feature of the language ensuring robustness and scalability. Overview of Java Threads is out of scope for this paper but relevant information on the topic can be found in Lewis and Berg [33], Oaks and Wong [34] or the official Java tutorials on concurrency [35], which were all used as reference during implementation.

The code was almost entirely implemented using Eclipse [36] IBM's Java based, open source Integrated Development Environment. The IDE was chosen because of its great flexibility and advanced testing and debugging tools. We used Holzner's Eclipse guidebook [37] as a reference.

3.4.2 Parallel Nonnegative Matrix Factorization (P-NMF)

The Parallel Nonnegative Matrix Factorization algorithm is based on the modified Pauca et al algorithm (see Sec 2.1.3). Since we are working with an SMP architecture, all the threads will share the data located in memory. However, each thread will only operate on selected bands in the image. We used spectral domain partitioning (see Sec 3.2) to evenly distribute the workload between the different threads (see 3.1).

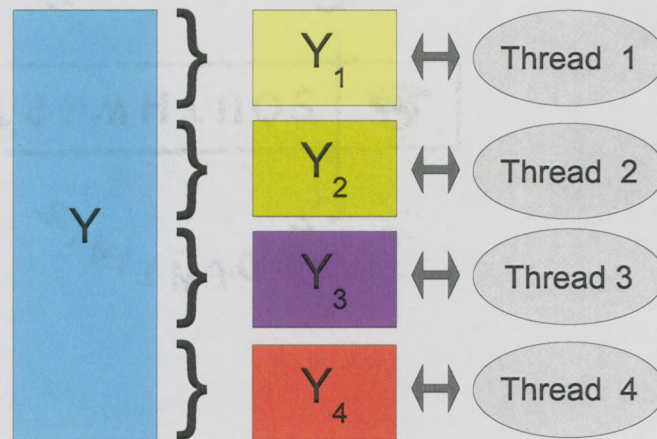


Figure 3.1: Data Distribution in P-NMF

New threads are spawned using the following code:

```
int start, end;

NMFThread[] tmp = new NMFThread[number_of_threads];

for(int i=0; i<number_of_threads; i++)
{
    start = i*(k/number_of_threads);
    end = (i+1)*(k/number_of_threads);
```



```

        if(tt == (number_of_threads -1))
            end = k;

        tmp[tt] = new NMFThread(start, end, times);
        tmp[tt].start();
    }

```

where `number_of_threads` is the actual number of threads to be spawned, and `k` is the same `k` that is defined in Sec 2.1.3. This way each thread will have roughly the same number of bands to work with. If the bands cannot be easily divided between the threads, then the last thread will usually have smaller amount of bands. The thread objects are collected in an array for future reference.

We have also introduced a noteworthy optimization to the original algorithm. Because we use the non-stochastic approach version H and W stay static during a single iteration. Therefore any computations involving only W and H could be computed once at the beginning of each iteration. We noticed that terms $W^T W$ and HH^T were computed inside one of the inner loops of the NMF code (see Appendix A). We have moved it outside of the loop. To speed up the process we decided to compute this step in parallel. Each thread will simply compute their assigned part of $W^T W$ and HH^T before entering the main computation loop.

Therefore the final algorithm has two parallel sections: one is a preprocessing step, and the other one is the actual NMF computation. Since the processing cannot start with an incomplete copy of $W^T W$ or HH^T the threads must synchronize after they all compute their respective preprocessing step. See Figure 3.2 for detailed diagram. The yellow squares represent sequential parts of the code, while the threads are marked in different colors.

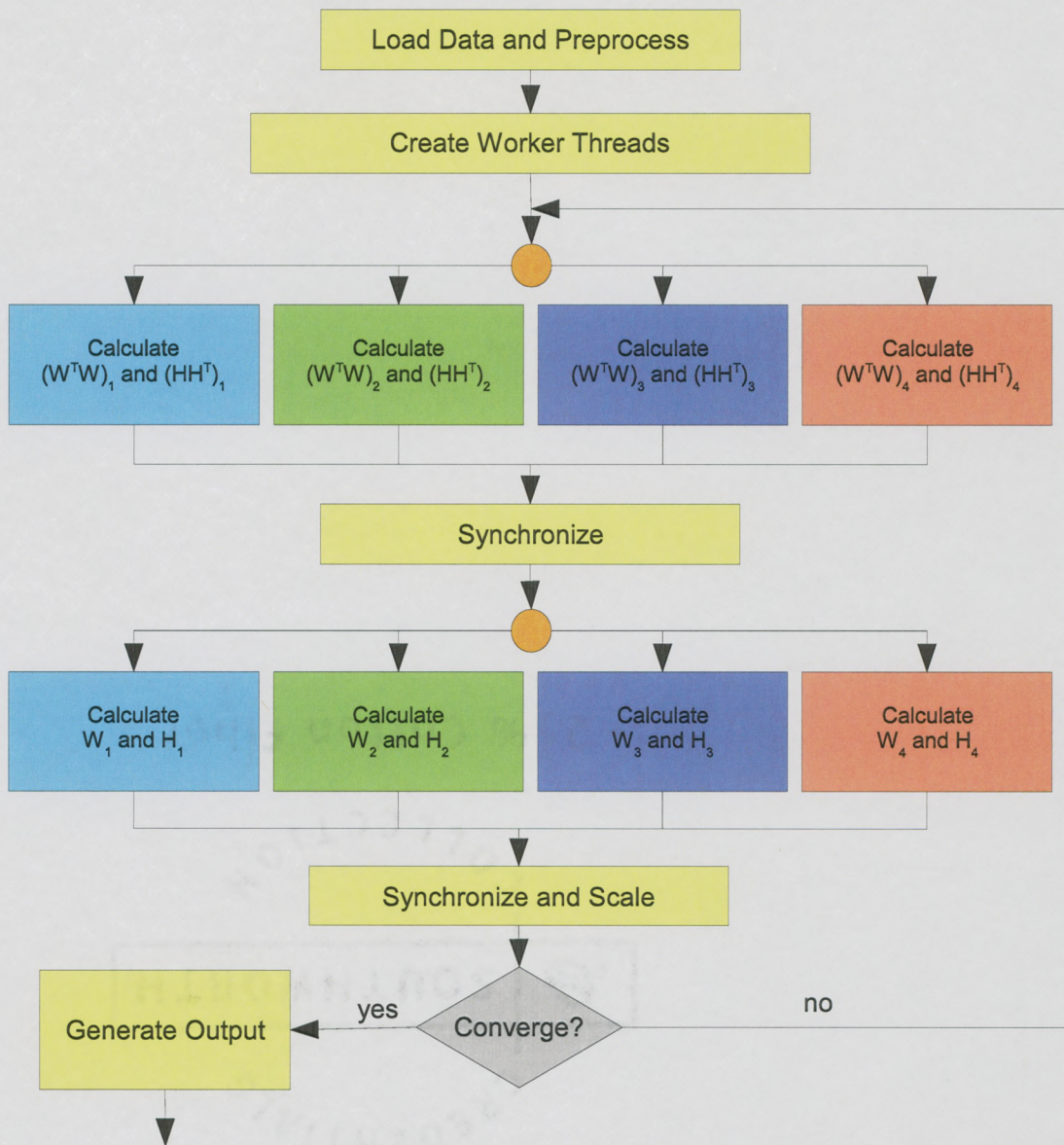


Figure 3.2: Diagram of P-NMF with 4 Threads

The computational complexity of the algorithm depends on the size of Y ($m \times n$), W ($m \times k$), H ($k \times n$) and the number of threads t . The complexity of the first parallel section can be computed as:

$$O\left(\frac{k}{t}kmn\right) = O\left(\frac{1}{t}k^2mn\right) \quad (3.2)$$

The cost of the second parallel section is:

$$O\left(\frac{k}{t}nm + nk + mn + mk\right) = O\left(\frac{1}{t}km(n+k) + kn(m+k)\right) \quad (3.3)$$

The formulas above only hold if we assume that the parallel speedup grows proportionally with the number of threads. The sequential steps do not contribute significantly to the complexity of the algorithm. The cost of scaling rows or columns of W or H is $O(mk+k)$ or $O(kn+n)$ while the test for convergence is $O(mnk+mn)$.

We have not extensively tested the results of this algorithm with respect to accuracy. However it can be noted that the code is semantically equivalent to the steps in the original algorithm, and our modifications do not modify its outcome. Semantic proof of this fact is beyond the scope of this paper. However we assume that this equivalence is true when presenting the experimental results.

3.4.3 Parallel Projected Gradient Nonnegative Matrix Factorization (PPG-NMF)

Parallelization of Lin's algorithm posed entirely different challenge than the standard NMF. Pucca's algorithm updated both H and W simultaneously in a long computationally expensive manner. It was relatively easy to divide that workload among different threads, and then collect the data at the end. PG-NMF on the other hand

uses much smaller iterative steps in which it calculates a new H or W, then tests it, and adjusts the α value, and re-calculates if needed. Furthermore, only respective H or W values change between these iterations, while the rest of data matrices stays unchanged. We were able to move most of the most computationally expensive calculations to a single preprocessing step which is executed outside the inner loop. This step is done only once for H and once for W per each major iteration ending in a test for convergence.

During our tests we noticed that the time spend doing the preprocessing is negligible compared to the time spent in the inner loop. One could usually count 10-15 or more iterations of the inner loop, in a single iteration of the outer loop. However, if we had used the spectral domain distribution method on H and W calculations in this algorithm, we would have to stop, and synchronize threads at each iteration of the inner loop, to evaluate and adjust α . The overhead of creating and stopping threads would be significant.

Instead we opted for a slightly different approach. Instead of parallelizing a single inner loop iteration, we decided to compute several of them at the same time. Thus, each thread receives it's own α value, which is used to find and test a new H or W value.

The α values are distributed among the threads using the following algorithm:

1. If we do not have a previous sum (ie. this is the first run):
 - (a) Initialize a single thread with $\alpha = 1$ and evaluate it
2. if we do have a previous sum S then:
 - (a) Let i be the number of threads such that $i \in \{1, 2, \dots, n\}$ where n is the total number of threads; Let e be an integer such that $e = 1$ if $S \leq 0$ or $e = -1$ if $S > 0$.

(b) Initialize all the threads with $\alpha = \beta^{ie}$

The first iteration is always performed sequentially. This might be a disadvantage, but in our experiments we have found that it is more beneficial to calculate the first step this way. Once we have our first resulting sum, we know whether to increment or decrement our α on the next iteration. The complexity of the parallel computations is:

$$O(k^2n + 2kn) = O(k^2n) \quad (3.4)$$

for H and:

$$O(k^2m + 2km) = O(k^2m) \quad (3.5)$$

for W. The performance gain in this scenario is not always consistent. On some iteration it might be more drastic than on the others. For example if the sequential algorithm would need d iterations to find the best possible α then in a perfect situation our implementation will need only $\frac{d}{n}$ where n is the number of threads because our algorithm is able to test n values at once. Unfortunately this is not always the case since many of values tested in parallel may be incorrect.

On the other hand, in the long run PPG-NMF should be able to close-in on an acceptable α value much faster than sequential algorithm. For example if we start at $\alpha = 1$ and the desired value turns out to be $\alpha = 10,000$ the sequential algorithms will need 4 iterations in our algorithm will only need two. On the first iteration it will test $\alpha = 1$ and determine that it needs to be increased. On the second iteration it will test 10, 100, 1,000 and 10,000 selecting the desired value.

Figure 3.3 shows a detailed diagram of the PPG-NMF algorithm. As before, the

parallel sections are in color, the sequential blocks are yellow, and logical branches are in gray.

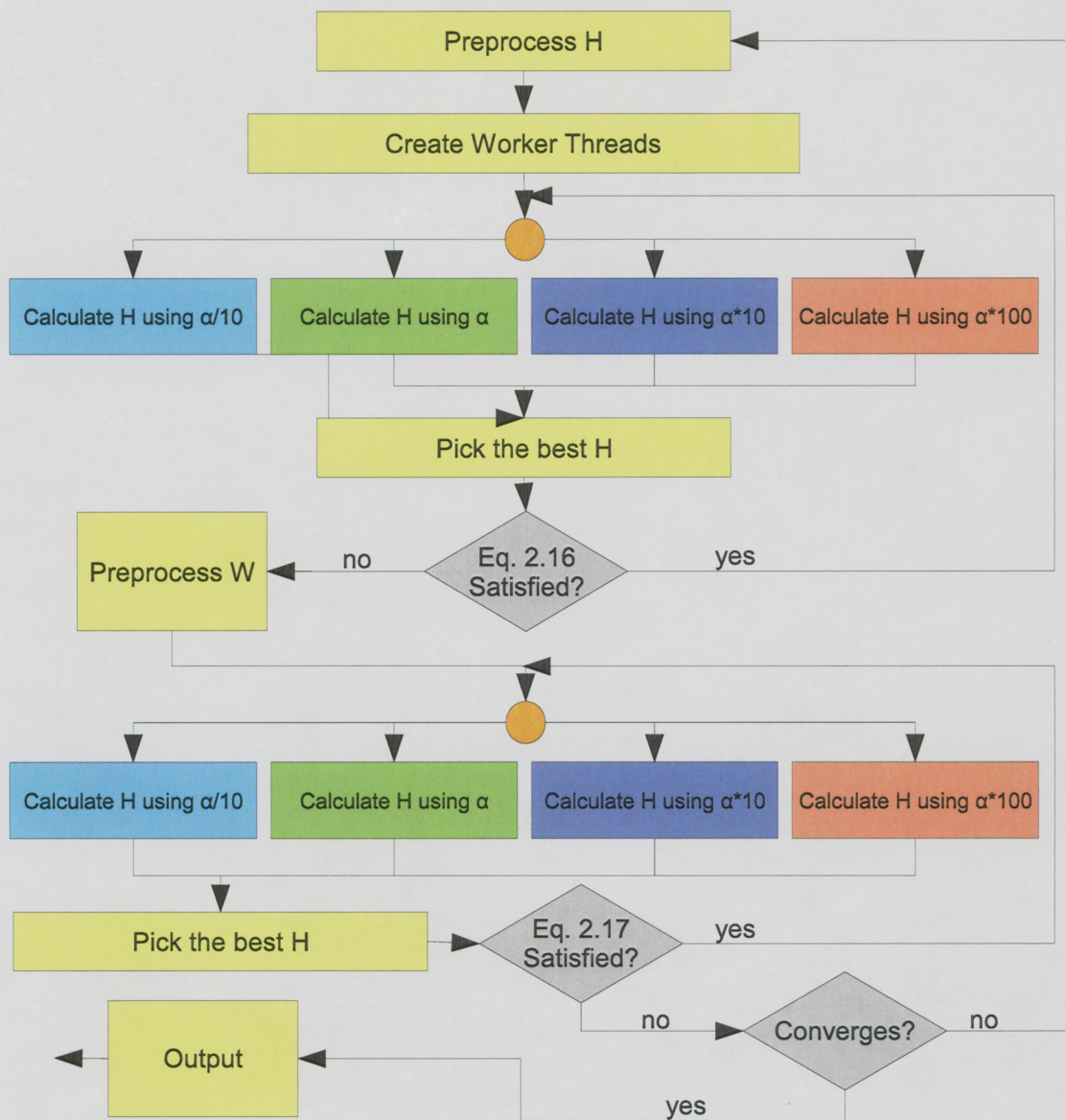


Figure 3.3: PPG-NMF Diagram

Chapter 4

Experimental Results

4.1 Experimental Data Set

We used two data sets in our experiments. Visualizations of both images are shown in Figure 4.1. The first sample (Fig. 4.1a) comes from the Hyperspectral Digital Imagery Collection Experiment (HYDICE) [38]. It is an areal photo of a foliage scene taken with a spatial resolution of 1.5m at wavelengths between 400nm and 2.5 micron. The data set uses sub-scenes provided by the Spectral Information Technology Application Center and has 85x185 pixels and 40 bands.

The second data sample (Fig. 4.1b) was taken using SOC 700 hyperspectral sensor currently available in the Remote Sensing Laboratory at Montclair State University. Originally the image was 160x160 pixels with 120 bands equally spaced within the 400nm and 900nm range (which encompasses the visible and near-infrared spectrum of light). For this experiment we have used 40 bands uniformly extracted from the image cube. The image depicts an artificial plant, in a light brown ceramic pot. Several leaves (shown in enhanced green on the picture) were placed between artificial leaves of the plant. The picture was taken outside on a sunny day to benefit from

the full spectrum illumination. The pot was placed on a large rock formation, with a brick wall forming the background.



Figure 4.1: Experimental Data: a) HYDICE data set - aerial shot of panels of various materials. b) SOC 700 data set with real and artificial vegetation.

Figure 4.2 shows the HYDICE sample with all the hidden panels highlighted.

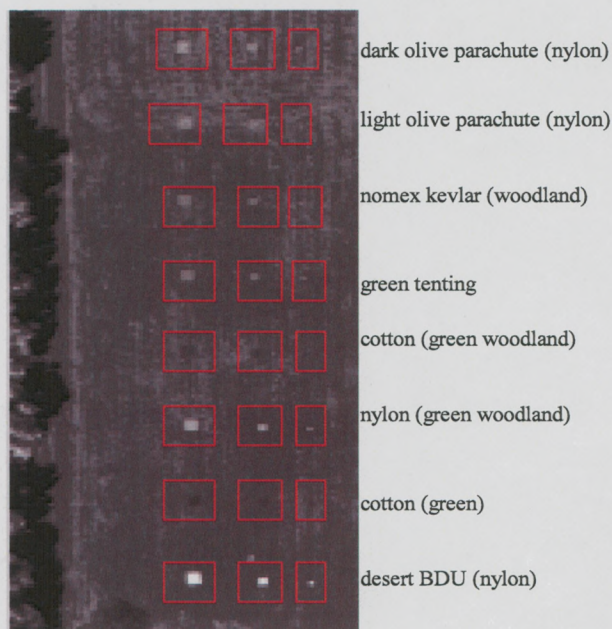


Figure 4.2: HYDICE Data Sample with the panels highlighted

Figure 4.3 shows the separated spectra obtained from the source image after stability was achieved.

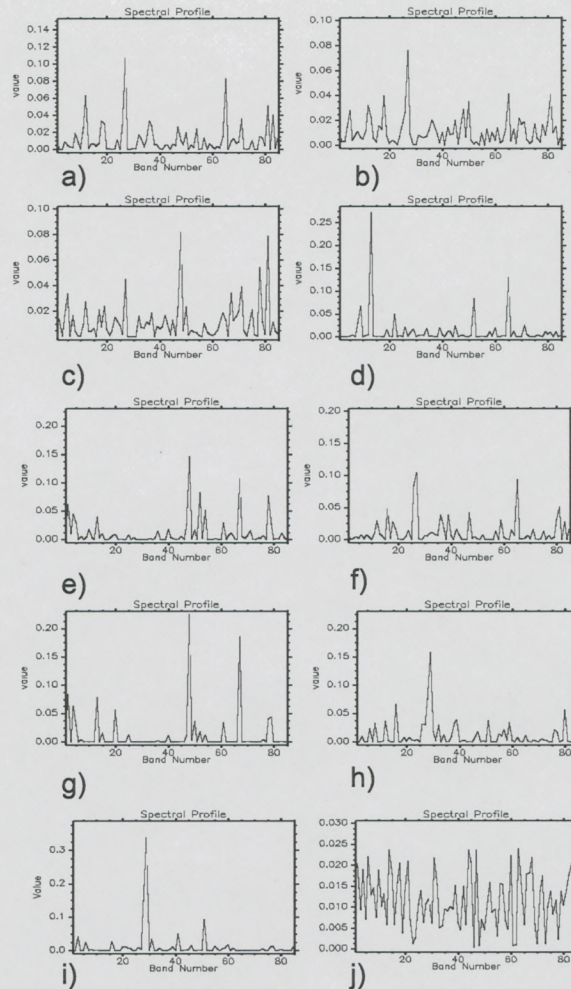


Figure 4.3: Abundance graphs for ten different materials present in the HYDICE sample after the stability was achieved. a) - h) panels in rows 1-8 (top down), i) vegetation patch, j) exposed ground

A similar separation graphs were observed for the SOC 700 sample (See Figure 4.4).

The same set of images was used in [10] and [9] producing very promising results in endmember extraction. The spectra graphs in Fig. 4.3 and 4.4 were obtained from these experiments. In this paper we will focused our discussion mainly on the speedup obtained thanks to parallel processing. For detailed discussion on the accuracy of the sequential NMF algorithms used to get these results we refer the reader to [10].

In section 4.4 we only discuss the speed and performance related issues. Both

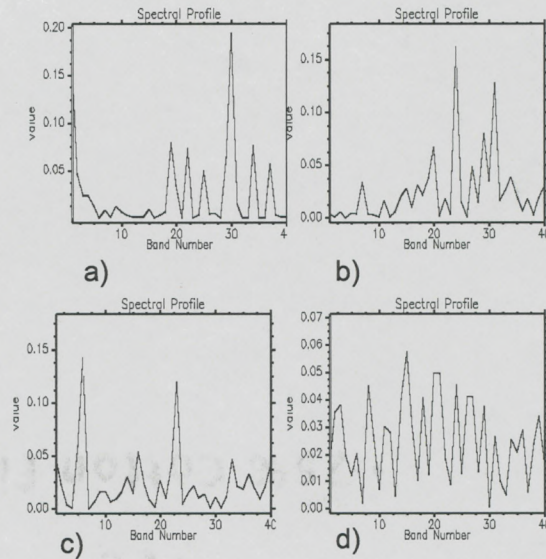


Figure 4.4: Abundance graphs for ten different materials present in the SOC700 scene after the stability was achieved. a) vegetation, b) artificial plant, c) ceramic pot, d) base rock

algorithms are semantically analogous to their sequential counterparts described and tested earlier, thus there was no need to collect their spectra. We believe that they would produce results very similar to the ones presented here.

4.2 Testing Platform and Machine Limitations

All the tests described in this paper were run on a SunFire v880 machine. The system has 4 UltraSparc9 processors clocked at 750MHz, 8 GB of RAM and is running Solaris 8. The code was run on Java 2 Runtime Environment, Standard Edition 1.4.0.1 (build 1.4.0.01-b03). This was a public university machine available to graduate and undergraduate Computer Science students and faculty. We had no control over system wide resource quota, and the tests were run from a standard student level account. The tests were ran from a shell script starting new processes automatically. We have tried to run the test during off hours where the system load was low, and few users were logged in. Each test would take several hours, and once the testing

started it would run through 5 complete sets before stopped. During that time there was no mechanism to check for system load or stop execution if some other users start resource intensive processes of their own. Therefore it is entirely possible that some of our results were skewed by outside interruptions caused by other people using the same machine. We collected several data sets and averaged them in an attempt to limit the impact of such outside influences on our experiments.

4.3 Testing Procedure

For each data set we conducted 5 tests using P-NMF and 5 tests using PPG-NMF. Each test consisted of independent 8 runs. The application was restarted and reinitialized after each run. Random matrices were initialized using a seed value. In total 5 randomly chosen seed values were used. Each one was assigned to one P-NMF test and PPG-NMF test. All the runs withing a given test were initialized with the same seed value so that they could be easily compared.

As our stooping criteria we have picked the change of $f(W,H)$. At each iteration the function was evaluated and compared to the result from previous evaluation. If the difference between the two was less than 0.001 the execution was stopped.

4.4 Results

In this section we will discuss the results of our tests. The following metrics were used to test our algorithms:

1. Accuracy - as the data is processed, at each iteration we evaluated the Equation 2.3 and stored the resulting value. We then graph this data against the number

of iterations to show how the value changes over time. The accuracy graphs are created by tracking 5 different sequential runs and graphing their results.

2. Average Execution Time - each run is timed using built in Java time functions. The average execution graph is created by averaging the execution times of all the runs in each test.
3. Average Execution Time per Iteration - we also graph the approximate execution time of each iteration. The graph is created by dividing the total execution time, by the number of iterations.
4. Speedup - finally, we also calculate speedup as defined in Section 3.2.1.

4.4.1 P-NMF

Figure 4.5 shows the accuracy progression of the P-NMF algorithm using the HYDICE data. Please note the the most of the significant changes in the value of Equation 2.3 happen during the first 100 iterations. After around 200 iterations the changes become very small. Choosing a better convergence criteria could potentially cut the total execution time in half or more. One should be careful however, since the graph shows very clearly that close to 100 iterations there are several "bumps" in the curve. These are local optima of the function which can cause the execution to end prematurely, with less than perfect distribution.

The SOC 700 tests show a similar pattern. The accuracy curve is much smoother, and even though the test ran for 2000 iterations, the change after the first 100 integrations was insignificant (see Fig. 4.6). If we use less strict convergence criteria, it would stop much sooner.

As it was, the sequential algorithm took approximately 45 minutes to finish. The best time was when ran with 4 threads, finishing around 23 minutes (See Fig. 4.7).

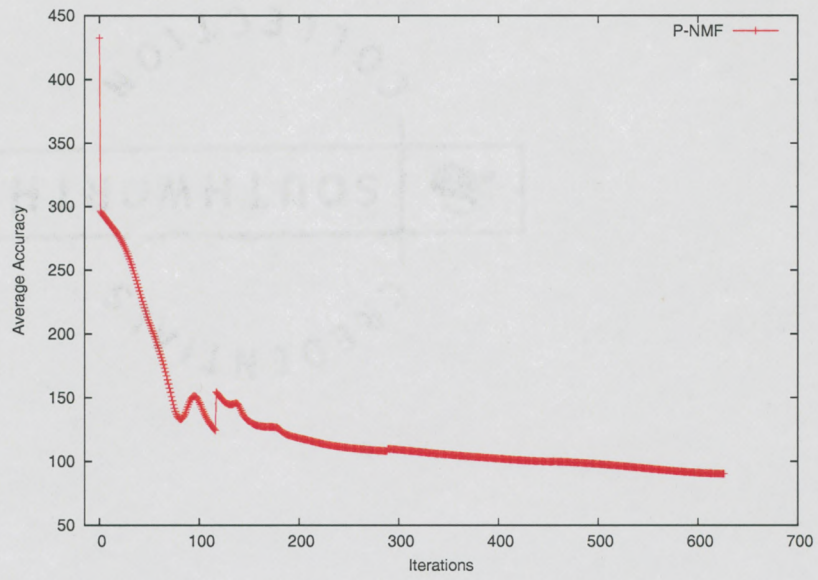


Figure 4.5: Accuracy Graph for P-NMF applied to the HYDICE data sample

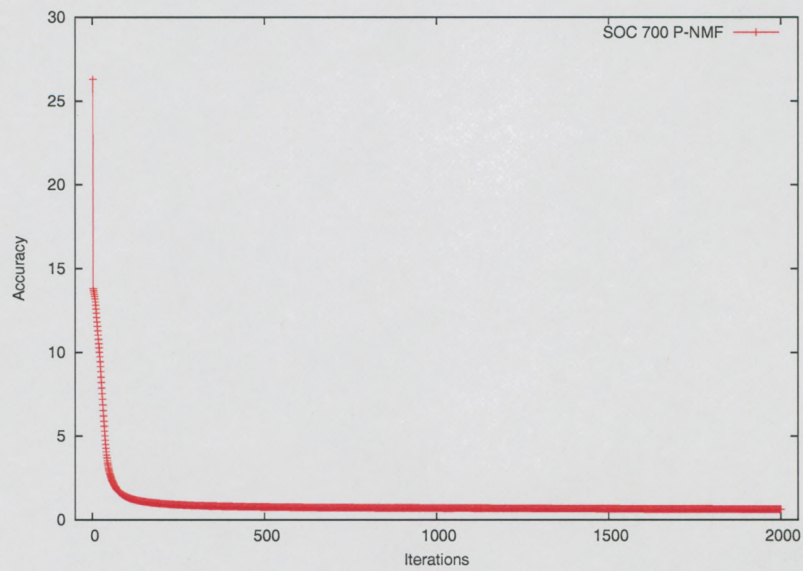


Figure 4.6: Accuracy Graph for P-NMF applied to the SOC 700 data sample

The SOC 700 sample took considerably more time due to its size. However the overall performance scaled very similarly to the HYDICE sample (See Fig. 4.8).

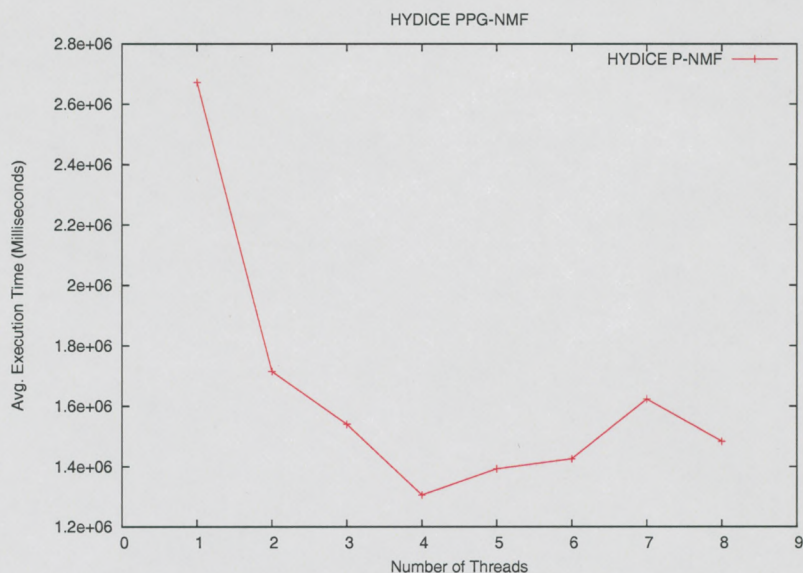


Figure 4.7: Average Execution Time for P-NMF applied to the HYDICE data sample

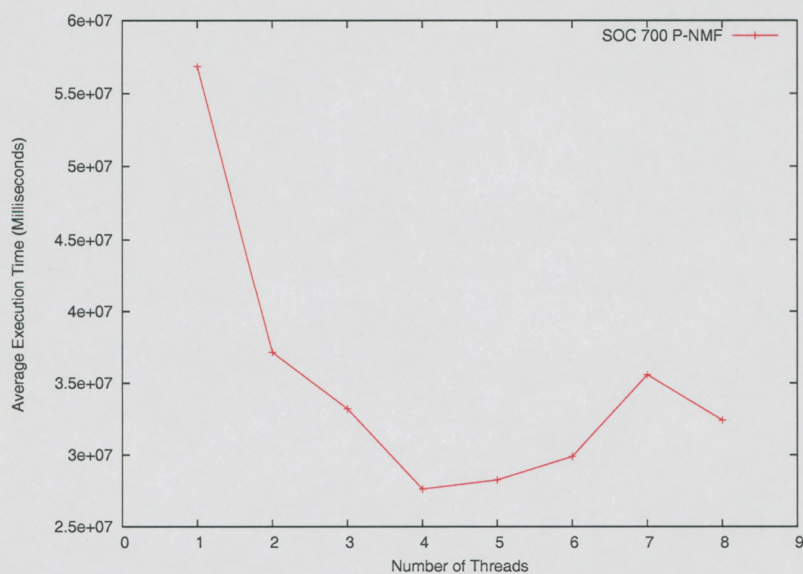


Figure 4.8: Average Execution Time for P-NMF applied to the SOC 700 data sample

You can also see that the average execution time per iteration is also drastically reduced in the multi threaded runs. (See Fig 4.9 and 4.10).

The average speedup values of P-NMF on HYDICE data set are present in Figure 4.11 and for the SOC 700 in Fig. 4.12. The parallel performance falls short of the

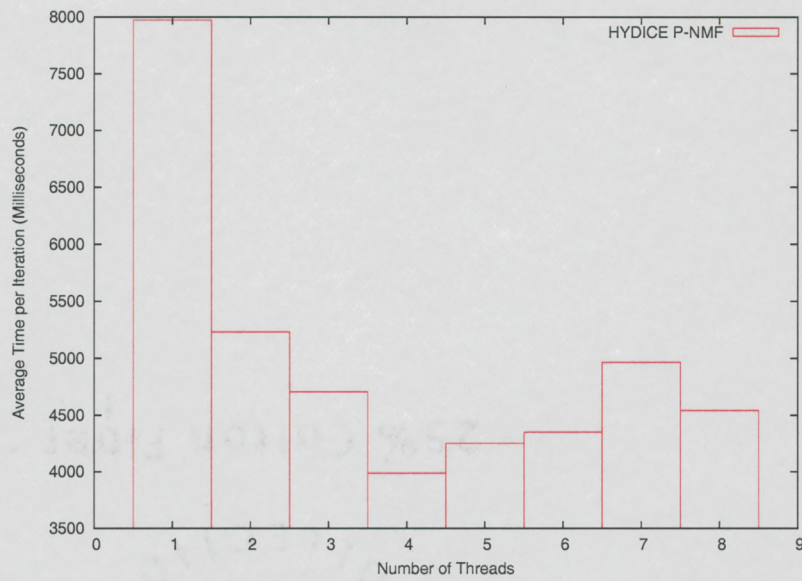


Figure 4.9: Average Time per Iteration for P-NMF applied to the HYDICE data sample

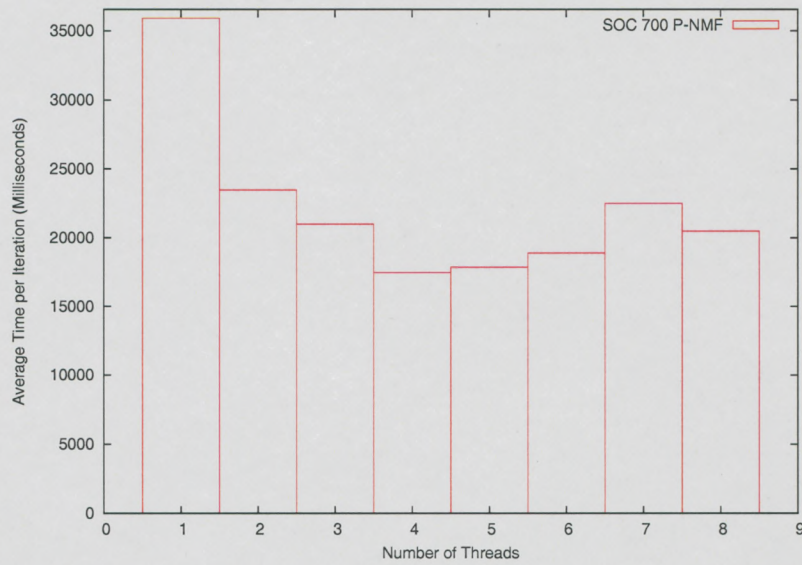


Figure 4.10: Average Time per Iteration for P-NMF applied to the SOC 700 data sample

ideal speedup but it is still a significant improvement over the sequential algorithm, since the best case scenario in our case manages to roughly cut the execution time in half.

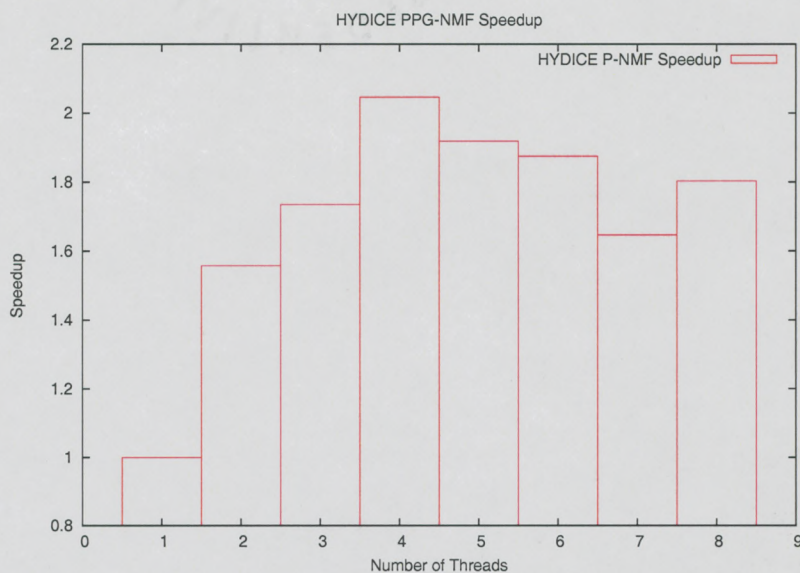


Figure 4.11: Average Speedup for P-NMF applied to the HYDICE data sample

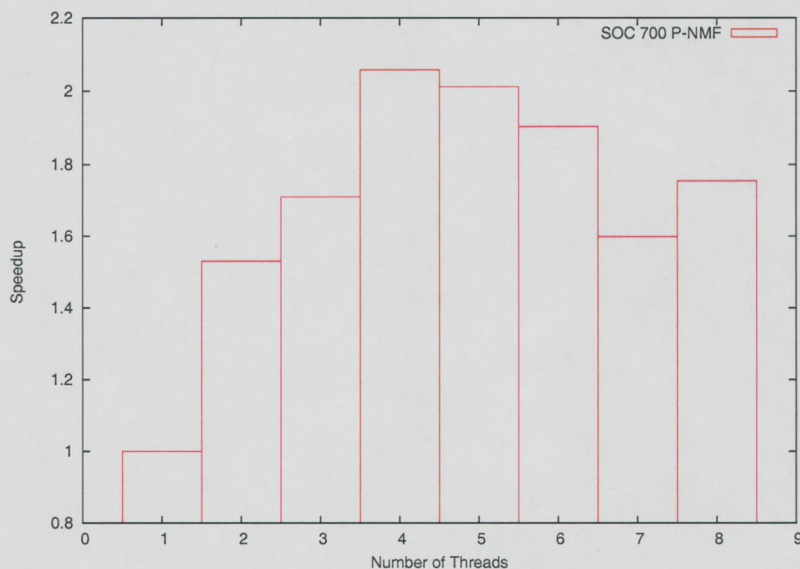


Figure 4.12: Average Speedup for P-NMF applied to the SOC 700 data sample

A perfect speedup, as described in Sec. 3.2.1 could not possibly be achieved because some parts of the algorithm must be run in sequence. Therefore at multiple occasions during the execution, the threads will stop, queue up or wait.

Since our test machine has only 4 CPU's available the runs with 5-8 threads performed relatively poorly compared to the 4 threaded run. Having more than 4 threads executed at the same time on a 4 CPU machine means that several threads will be assigned to a single processing unit, and will have to be swapped in and out at equal intervals. Thus the speedup is always dictated by the number of available processors.

As shown in the graphs above, the performance of P-NMF is directly proportional to the number of available CPU's. It can be expected that the algorithm will scale very well when run on machines with more than 4 CPU's. A workstation with 8 processors for example should be able to cut down the execution time up to 4 times.

4.4.2 PPG-NMF

The accuracy curves for PPG-NMF are very similar to the P-NMF ones. It is worth noting that unlike P-NMF, PPG-NMF does not seem to have very pronounced local optima. Instead it seems to decrease really fast in the first 10 iterations, then hit a small plateau and, drop again and then decrease very smoothly for the rest of the run. After 50 iterations changes are very minuscule. (See 4.13)

The PPG-NMF finishes processing the HYDICE sample in roughly 12 minutes. The multi-threaded runs show some improvement, but it is not as large as in case of P-NMF. Using 4 threads we can shave off over 2 minutes from the total execution time. (See Fig. 4.14 and 4.15)

The calculated speedup values show that the 4 thread algorithm is only 1.14 times better than the sequential one. (See Fig. 4.16). We will attempt to explain the difference in speedup between the two algorithms in Section 4.4.3.

The accuracy curve for PPG-NMF with the SOC 700 data set is very similar to

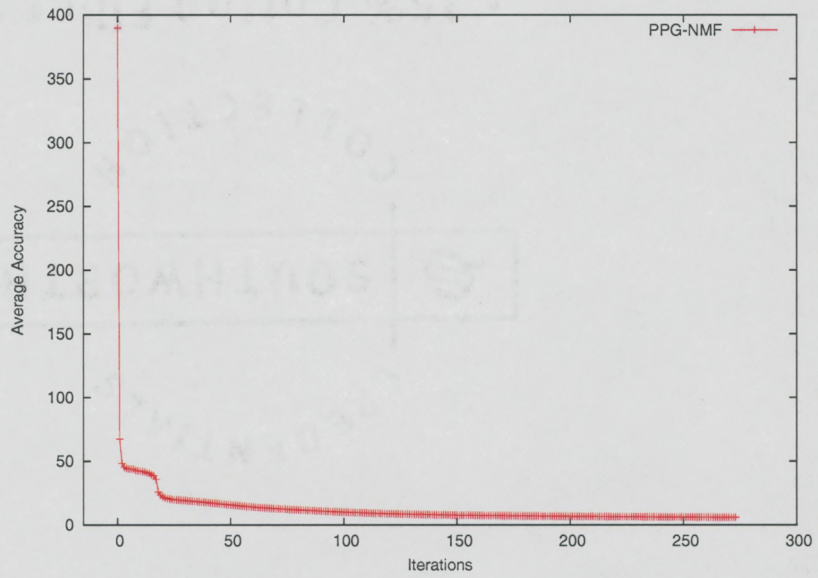


Figure 4.13: Accuracy Graph for PPG-NMF applied to the HYDICE data sample

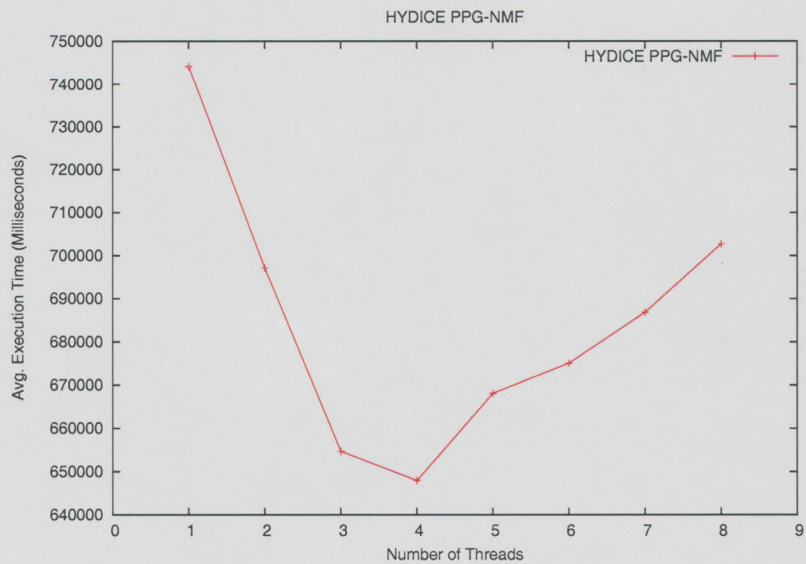


Figure 4.14: Average Execution Time PPG-NMF applied to the HYDICE data sample

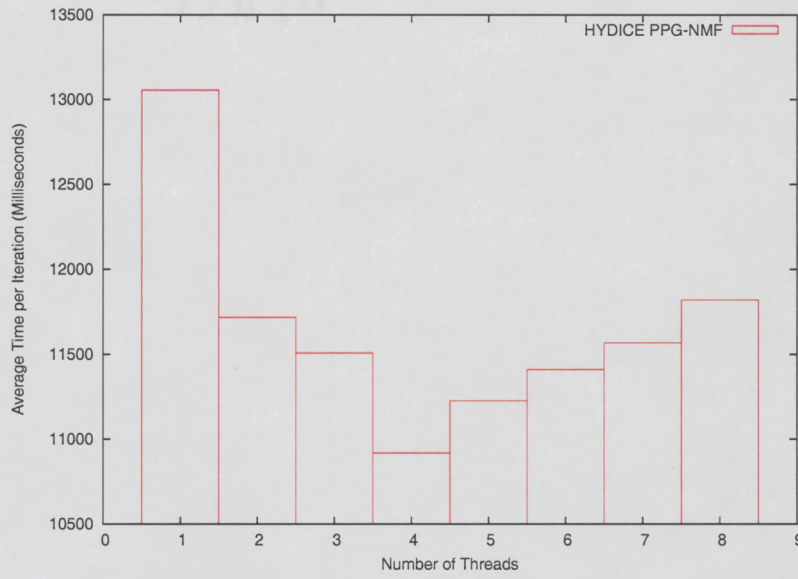


Figure 4.15: Average Execution Time per Iteration PPG-NMF applied to the HYDICE data sample

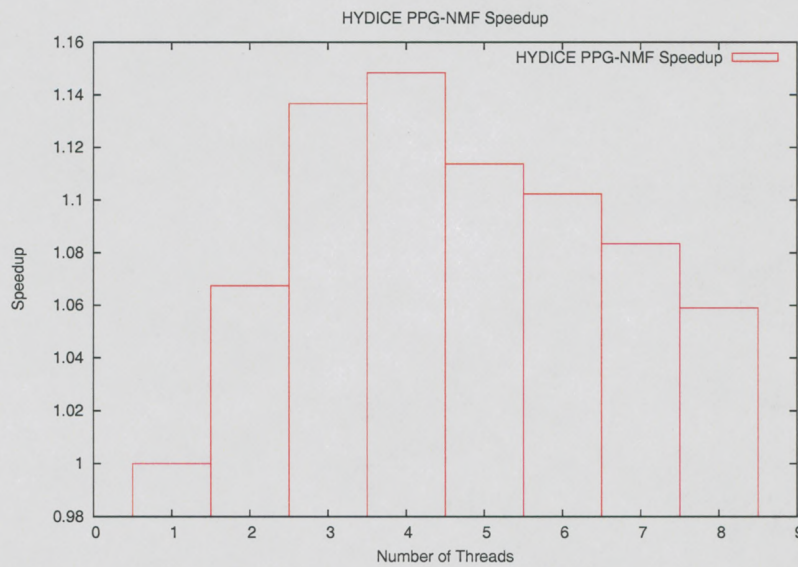


Figure 4.16: Speedup for PPG-NMF applied to the HYDICE data sample

the HYDICE one. In both cases the function 2.3 stops changing rapidly after roughly 50 iterations.

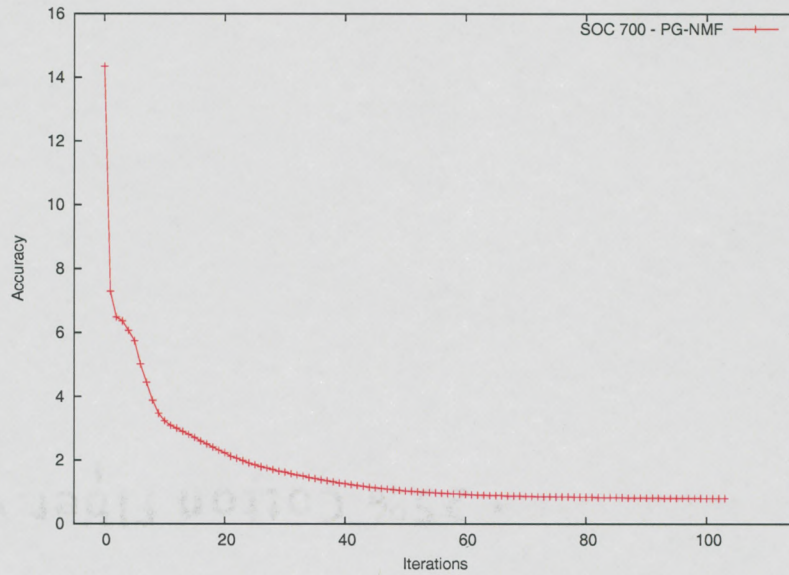


Figure 4.17: Accuracy Graph for PPG-NMF applied to the SOC 700 sample

Plotting the average execution time (Fig. 4.18), and average execution time iteration (Fig. 4.19) shows a strange anomaly. The run executed with 3 threads ends up being slightly slower than both the 2 and 4 thread runs. It is worth noting that no such phenomena was observed with the HYDICE data set (See Fig. 4.14).

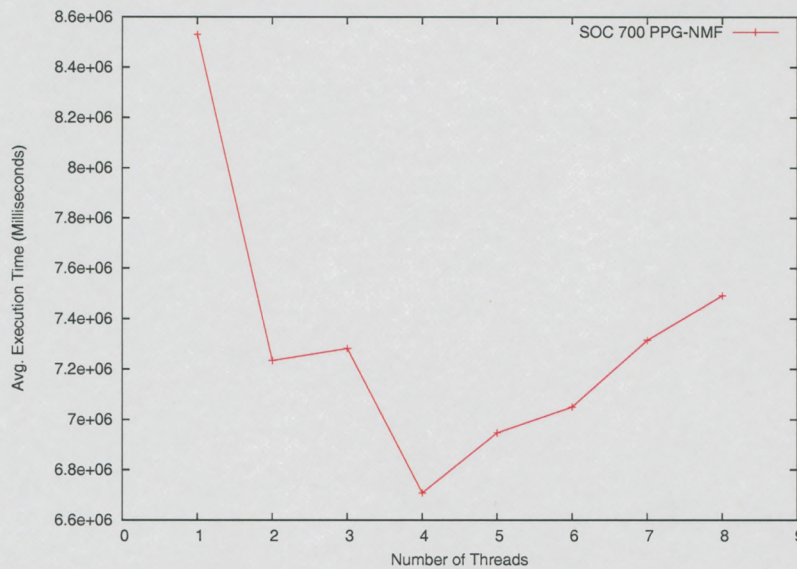


Figure 4.18: Average Execution Time for PPG-NMF applied to the SOC 700 sample

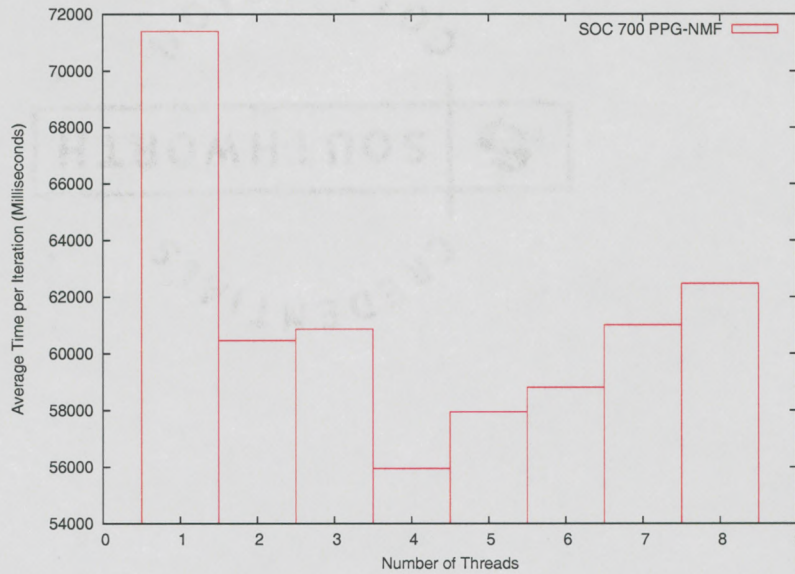


Figure 4.19: Average Execution Time per Iteration for PPG-NMF applied to the SOC 700 sample

The speedup achieved with the SOC 700 data was slightly better than the one observed with HYDICE. The run with 4 threads ended up being 1.27 times better than the sequential one.

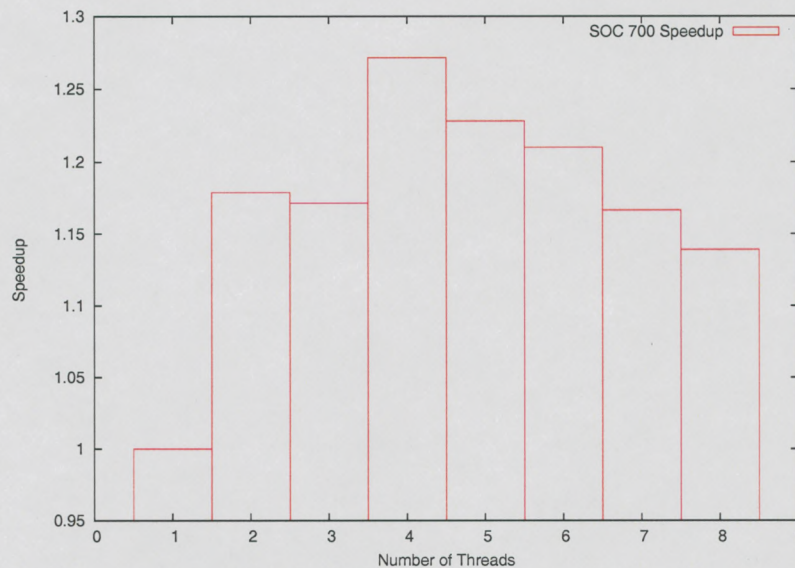


Figure 4.20: Speedup for PPG-NMF applied to the SOC 700 sample

This might seem like a very poor result, but in fact this is a significant improvement with respect to actual processing being done. For example, let's examine how many times on average does the PPG-NMF algorithm have to re-evaluate α on any given

iteration. Let's assume that a PPG-NMF "round" consists of evaluating Equations 2.16 and 2.17 and testing their results using Equations 2.18 and 2.19. If the test fails, we need to go back and calculate a new H or W with a modified α .

In our sequential runs we essentially saw on average 5.5 rounds per iteration. For example, it would take 3 – 4 rounds to find H and 5-6 rounds to find W. Figure 4.21 shows that the ratio of rounds per iteration drops significantly as we add more threads. This ratio is almost identical in the HYDICE sample. With two threads we only require an average of 3 rounds per iteration. With 4 threads this ratio drops to 1.5 – 2. With seven or more threads, we always get one round per iteration. This means that the correct α was always found among the 7 increment steps above or below the starting point.

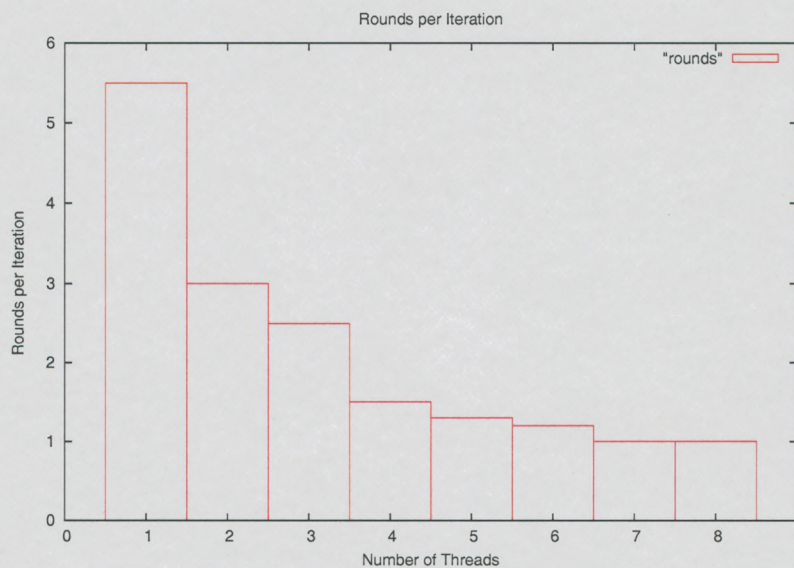


Figure 4.21: Rounds Per Iteration in PPG-NMF (SOC 700 sample)

One explanation of the poor speedup can be the fact that 4 threads we still occasionally have to do more than one round per iteration. Therefore, our parallelization scheme was not being used up to full potential. To get the best performance we needed at least 6 or 7 threads. Of course our experiments with 5-8 threads performed poorly, because of hardware limitations. With only 4 CPU's available the threads

were competing for resources. Given 8 processors PPG-NMF could potentially show much better speedup results because.

It is worth noting that executing PPG-NMF on a machine with more than 8 CPU's will not impact it's performance since the efficiency of the algorithm is largely dependent on rounds per iteration. It is not possible to further improve this ratio beyond the 1 round per iteration average. The peak performance of PPG-NMF can be achieved with factorization dedicated 8 CPU machine running with 8 threads.

4.4.3 Discussion of Results

As can be seen from the results above, PPG-NMF is on average faster than P-NMF. However, P-NMF is better suited toward parallelization. The multi threaded P-NMF implementation will scale well as more CPU's and threads are added making it a very attractive choice for high end, multiple CPU machines. It can also be ported over into clustered.

On the other hand, PPG-NMF doesn't scale that well. There seems to be an upper cap on how fast the algorithm can go, beyond which adding more processors would be wasteful. However, since it was on average able to process the same data sample in roughly half the time as PPG-NMF it might be a good alternative for the low and medium range multiprocessors. The optimal performance seems to be achieved at around 8 CPU's.

This implementation is not appropriate for clusters. However an alternative implementation which distributes data in a different way could potentially change this.

Chapter 5

Java Based Hyperspectral Image Processing Toolkit

5.1 Introduction

To perform any analysis or feature extraction, it is necessary to implement tools that are able to manipulate the hyperspectral data. The integral part of this thesis was developing a Java based hyperspectral processing framework with a graphical user interface. This posed several challenges that needed to be overcome before a workable implementation could be created.

The framework needed appropriate data structures to store, and process hyperspectral files. A graphical user interface including image visualization tools were needed to display processed image on the screen either in gray scale or in color. Other auxiliary tools were also needed in order to allow users to obtain a data dump of the image, or inspect the spectra values at a given location. Finally, a complex input/output system had to be designed to handle the wide variety of hyperspectral data formats available.

5.2 Data Structures and Representation

In Section 1.2.2 we explained the complexity of hyperspectral files. One of the most fundamental problems when developing algorithms for processing such data, is coming up with data structure which would support their complexity. What are the requirements for such a data structure?

1. ability to store multidimensional data
2. support for large number of data types
3. compatibility with Java visualization tools

Since java does have limited support for rendering raster images we chose to extend that functionality to include handling hyperspectral data. The `DataBuffer` class in the `java.awt` is used internally by several of the raster image implementations. It is an abstract container class designed to hold a set of arrays (bands). Each band is composed of a set of scan lines which can be accessed individually [39].

Usually only 3 bands are used for red, green and blue channels but there is no limit to how many bands can be defined. Furthermore, `DataBuffer` class itself is abstract. There is a dozen of implementations of this class, each designed specifically for a certain data type (ie. `DataBufferDouble`, `DataBufferInt`, `DataBufferByte` and etc..). While not all possible data type implementations are present in the standard libraries, new ones can be easily added by extending the `DataBuffer`.

In our application all the hyperspectral data is internally stored in data buffers. When reading in, the data format, be it BIL, BIP or BSQ is "unrolled" into the standardized format described in Section 2.1.2, and then fitted into appropriate `DataBuffer` implementation.

Since the underlying structure of `DataBuffer` is a two dimensional array, we sometimes choose to extract it for performance reasons. During the iterative processing of NMF, interfacing `DataBuffer` via method calls would create unnecessary overhead. Thus the processing algorithms work with raw arrays, usually of type *double*. When processing is done, data is once more wrapped into a `DataBuffer` before being returned to caller.

Nearly all the modules, save for those which actually perform I/O operations, only need to know how to read from, and write to the generic `DataBuffer` class. This provides a unified application-wide data interface that makes it easy to write additional modules, or since `DataBuffer` is part of the core AWT API, to share data with other Java imaging applications.

5.3 Designing a Graphical User Interface

The discussion of GUI design should start by choosing appropriate widget toolkit. Java offers developers several attractive graphical environments. We will first take a closer look on each of them. Visualization details will then be explained in terms of that chosen toolkit.

5.3.1 Choosing a Widget Toolkit

There are three competing major widget toolkits that can be used for building graphical user interfaces in Java: AWT, Swing and SWT. Both AWT and Swing are currently included in Java Standard Edition, while SWT is an external set of libraries that needs to be downloaded and deployed separately.

The AWT toolkit was the first widget system implemented for Java. It is composed

from fairly simple wrapper classes which in turn make calls to the native windowing environment to display GUI elements. Since each platform has a different set of native widgets, Sun only included the most basic ones in AWT [37].

Swing toolkit was developed in 1998 as a replacement of AWT. It abandoned the idea of using the systems native windowing environment, and designed a complete, feature rich widget toolkit entirely in Java. Swing is still in active development, and offers an impressive number of different graphical tools and GUI elements to the developers. For example, the package included in Java 1.4 includes 85 public interfaces, and 451 public classes [40].

The pure Java implementation however has proved to be both a blessing and a curse for this toolkit. Since Swing elements are really Java objects, they can only communicate with the underlying operating system through the JVM. In some cases this can create significant overhead, and thus Java based interfaces will often appear to be less responsive, or slower than their native counterparts [37].

The Standard Widget Toolkit (SWT) [41] developed by IBM is a hybrid between Swing and AWT. It combines both approaches by including both calls to native widget elements, as well as implementing pure Java based ones. This helps to significantly improve the performance without sacrificing any of the advanced features one may look for in an enterprise grade product [37].

The difference in performance, and responsiveness between Swing and SWT however is widely disputed and highly controversial. For example some benchmarks claim that Swing can outperform it's competitor with respect to speed of rendering and redrawing windows on non-windows platforms [42]. Thus it is not very clear if SWT is really faster than Swing, or if overhead of the native calls and communication between Java and non-Java elements diminishes any performance gains stemming from using a native widget implementation.

Benchmarking the two toolkits is out of scope for this paper, and thus we chose Swing as our GUI toolkit because it is a Java standard, and it reduces the complexity of our code by eliminating dependency on a third party library.

5.3.2 GUI Overview

When designing a graphical user interface for our framework, we wanted to emulate some of the existing graphical suites such as Photoshop, by using a single program window. Since our application would usually require several different graphical modules to be on the screen at the same time (image viewing pane, processing toolbar etc..) we wanted to bind them all to a single system window which would be easily minimized or closed. This is different from the approach used in graphical suites such as the open source photoshop like GIMP suite or the popular hyperspectral processing suite called ENVI. These applications employ several free floating windows, each of which has to be closed, minimized or repositioned separately.

The main program window is built using the Swing class called JDesktopPane. It creates a single window, which can contain several sub windows that are completely contained within it (ie. can't be detached, or moved outside of the main window). See Figure 5.1.

Since the hyperspectral images are usually simple binary files with no headers, we do not attempt to guess the type and format of the file upon opening. Instead we pass that responsibility to the user. Identification of data type and file format based on analysis of raw binary data is a very complex problem that is out of scope for this paper.

The Open dialog allows the user to input important information about the files such as it's width and height in pixels, number of bands, data type and hyperspectral

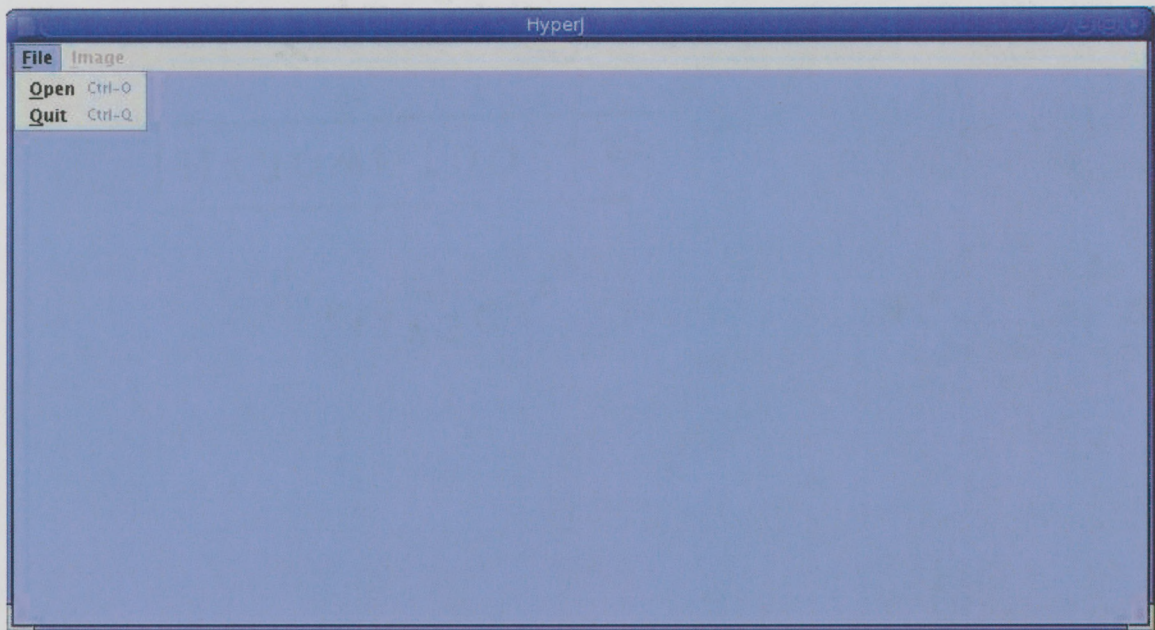


Figure 5.1: HyperJ GUI: Main Window

encoding format. It also allows the user to browse and choose the file he wants to open (see Figures 5.2 and 5.3).

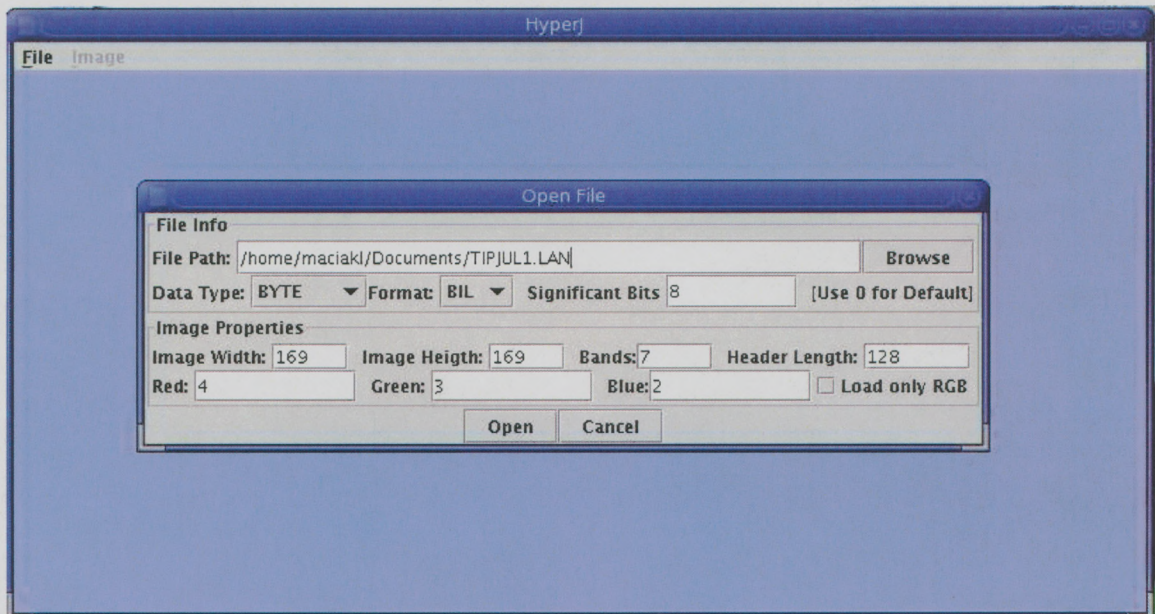


Figure 5.2: HyperJ GUI: Open File Dialog

The user input will be automatically saved into a file `.hyperj` located in the user's home directory. On windows machines this would usually be:

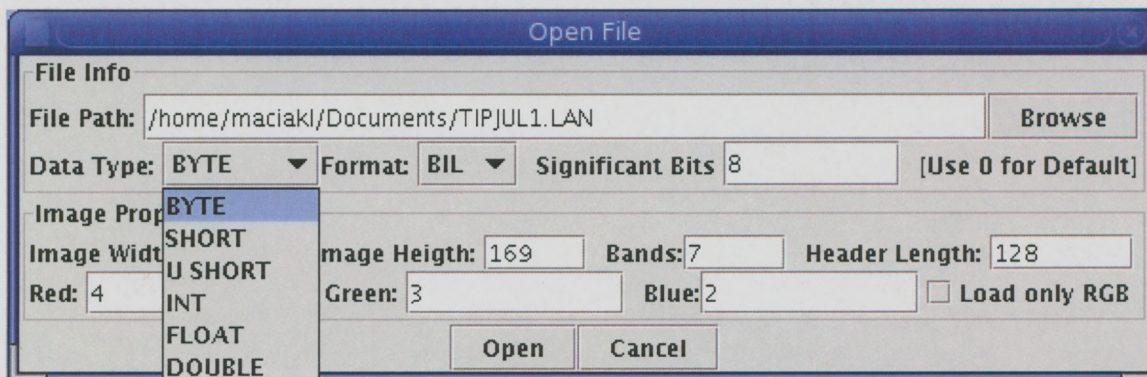


Figure 5.3: HyperJ GUI: Open File Dialog Details

C:\Documents and Settings\[Username]\Application Data\.hyperj or
%appdata%\.hyperj

On Linux and Unix on the other had it would be:

/home/[username]/.hyperj or
~/hyperj

It is a very simple text file formatted according to the "key=value" formula that is standard for java Properties. On subsequent runs, the application automatically reads in the file, and fills in the data. Thus the user can easily go back and reopen last viewed file without the need of retyping all the information in the Open File dialog. The .hyperj file is also used in the command line mode (see Section 5.5). Please note that under Linux and Unix this will be a hidden file, but under Windows it will be a normal file located in a hidden directory.

Once the file is open, the user will see the Image Toolbox (See Fig: 5.4). It includes some information on the file and a set of function buttons that allow the user to display the image on the screen, or show a textual representation of the image contained in each band on the screen. It also allows to initiate one of the NMF processing algorithms.

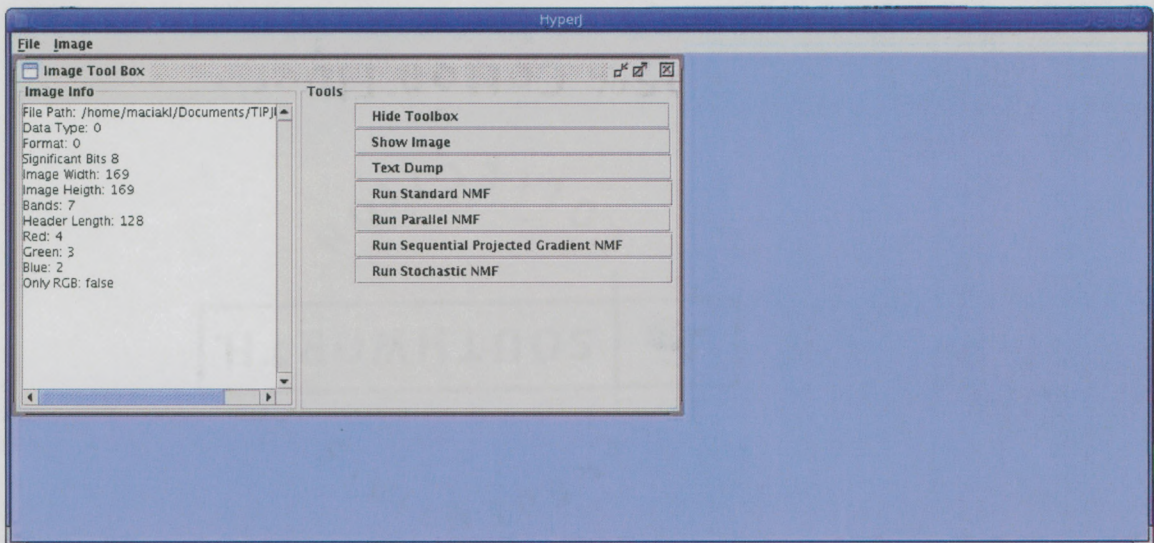


Figure 5.4: HyperJ GUI: Image Toolbox

Figure 5.5 shows the application in action. The opened hyperspectral image can be seen on the right, and the text dump of the bandvector data is visible below the toolbox.

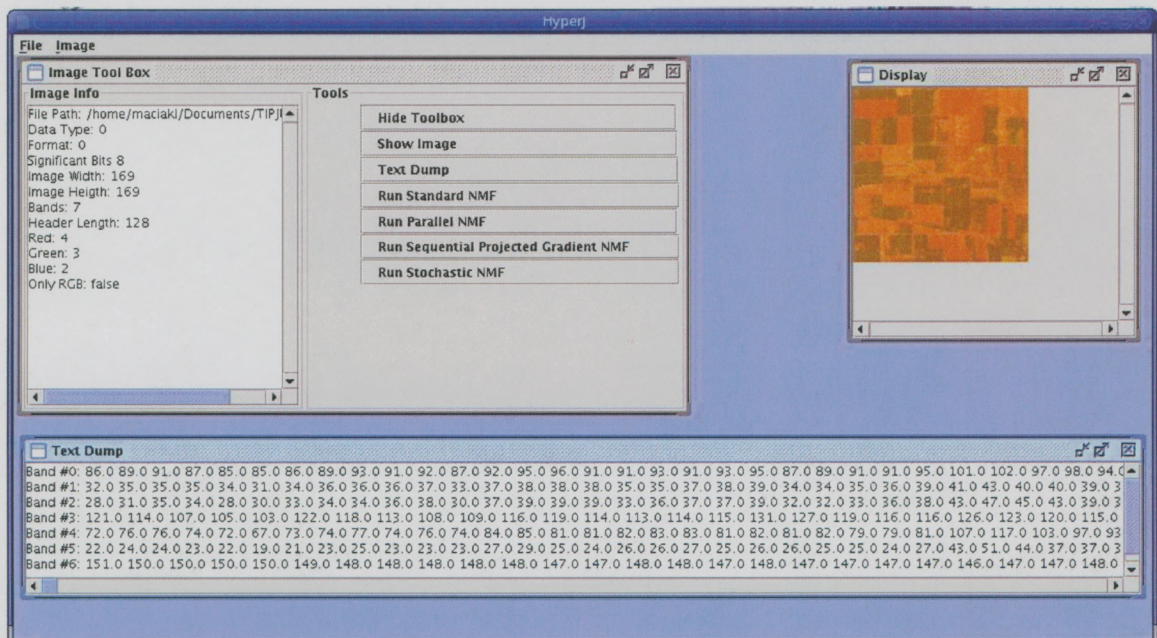


Figure 5.5: HyperJ GUI: HyperJ with an open image and text dump

At the present, only one image can be open within the GUI at a time. When making the decision regarding opening images, we were mostly concerned about per-

formance and memory usage. Holding several large images in memory would easily degrade the overall performance, and interfere with the NMF algorithms. However, the code could be easily modified in the future to allow multiple images to be open at the same time.

5.3.3 Visualization of Hyperspectral Data

In Java 2D API the basic representation of an image data is the `Raster` class. `Raster` is a high level abstraction which encapsulates the raw image data (as a `DataBuffer`) and a `SampleModel` object which describes how that data is organized. For our purposes we used `BandedSampleModel` implementation which accurately represents multi-band data and which can be easily generated based on the `DataBuffer` and known properties of the image [39].

Unfortunately `BandedSampleModel` does not implement all of the required datatypes. The only available types are byte, unsigned short, short, integer, float and double. We are planning to extend this class and add implementation for unsigned integers, bytes and long integers at some point in the future. In the meantime however, our application was designed to handle only the `DataBuffers` with the known and supported types [39].

In order to display a `Raster` on the screen, we must combine it with a `ColorModel` object which encapsulates methods for translating a pixel value to color components (red, green and blue) and an alpha component. Raw pixels contained in the `Raster` must be transformed to these components before they are rendered. Similarly to `SampleModel` the `ColorModel` only supports the same 6 basic data types and will at some point need to be extended.

Given a `Raster` and a `ColorModel` one can generate a `BufferedImage` object

which is an actual renderable element. It can be displayed on using any Java tools support rendering images. For example, in the `maciak.hyper.gui.ImageViewer` class we use `JLabel` as he canvas on which out image is displayed.

The `maciak.hyper.img.SpectralImage` class aims to simplify this rather complex environment. It can be initialized with a `DataBuffer` object, and few other attributes (passed in as integers, or predefined constants) and then automatically generate a `SampleModel`, `ColorModel`, `Raster` and `BufferedImage` as needed. Figure 5.6 illustrates the relationship of the `SpectralImage` class to built in elements of Java 2D API.

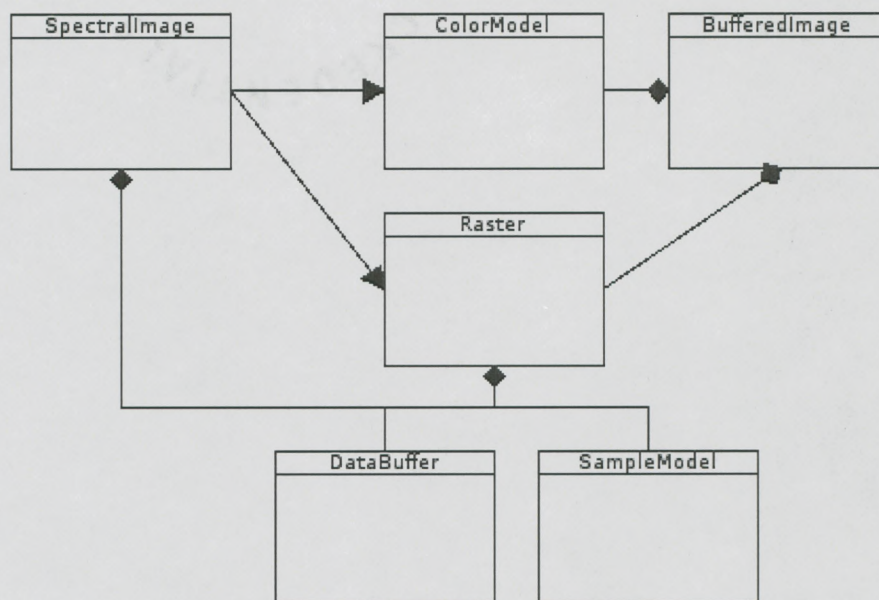


Figure 5.6: Visualization of Hyperspectral Images using Java

5.3.4 Starting the GUI Interface

To start the GUI Interface you need to run the Main class located in the `maciak.hyper` package. To accommodate the memory requirements of this program you may need to increase the maximum heap size for the virtual machine. In our tests we used the following line to start the program:

```
java -Xmx1024mb maciak.hyper.Main
```

We recommend using at least 1GB of heap memory. When using the pre-compiled compressed JAR file version of the tool the start command would look as follows:

```
java -Xmx1024mb hyp.jar
```

Please see Appendix A for instructions on how to compile the code and create the JAR file.

5.4 Handling I/O

Choosing a unified data exchange format to be `DataBuffer` as outlined in Section 5.2 is only part of the solution. The input and output algorithms for a hyperspectral processing toolkit must be able to handle the variety of different image encodings (see Section 1.2.2). Following steps have been taken to implement a working I/O framework for our application.

5.4.1 Unified Framework for Handling Different Hyperspectral Data Types

We have developed two base classes called `SpectralReader` and `SpectralWriter` as part of the `maciak.hyper.io` package. These two methods create general framework for handling I/O data, and include abstract methods such as `readFully()` or `writeFully()` that need to be implemented by child classes. Since hyperspectral images use so many different data types and encodings creating a generic logic for read and write methods is not possible.

Thus, next we developed a set of child classes which implement the abstract read and write methods for different data types. In total, we need up to 42 different child classes to fully support all the possible combinations of data types and encoding formats. Following convention was adopted to name these classes:

$$\langle EncodingFormat \rangle \langle DataType \rangle [Reader|Writer]$$

For example: `BSQDoubleReader`, `BILIntReader` or `BILUnsignedByteWriter`.

The `SpectralReader` and `SpectralWriter` classes also have a static methods called `getReaderByType()` and `getWriterByType()` respectively. Both methods take in a `ImageFile` object which is also part of the `maciak.hyper.io` package. This object is used for storing information about an image file on the disk. As opposed to `maciak.hyper.img.SpectralImage` it does not contain any actual data in it. What it does contain is information and methods directly related to I/O such as image header offset, line length, data type, encoding format and etc.. It can also be used to generate blank data buffers of appropriate type.

Based on the information stored in the `ImageFile` object `SpectralReader` and

`SpectralWriter` create and return a reader or writer object of given type back to the caller. However, the return type of the get methods is generic `SpectralReader` and `SpectralWriter` object. This is intended polymorphism ensuring that outside classes only have to deal with a single interface for reading and writing, even though there are actually 42 different low level implementations of the read and write methods.

There is a higher level of abstraction built into the system. Most of the display, and GUI based classes never interact directly with the reader and writer classes or the `ImageFile` class. All of them simply interface with static methods of the `ImageFactory` class from the `maciak.hyper.gui` package. This class takes a set of parameters, and then simply returns a `SpectralImage` object.

Figure 5.7 shows a partial class diagram for the I/O framework, showing the relationships between classes involved in reading hyperspectral images. Only two implementations of `SpectralReader` were shown in the image, to avoid visual clutter (including 42 nearly identical classes would add unnecessary complexity to the diagram). Similarly, writer classes were also omitted. They they have the same kind of relationship with `ImageFactory` as the reader classes.

5.4.2 Big Endian/Little Endian Conversion

Java presumes that binary data is stored most significant byte first. All the files written by java are in Big Endian notation. When reading data from a file Java will usually assume that it is in Big Endian notation as well. This is not necessarily an issue when reading files using basic stream reader applications, byte by byte. In such situation one is free to choose the reading order. However at times it is convenient to harness the power of Java core libraries to streamline, and automate mundane I/O functions. For example `java.io.RandomAccessFile` provides much more convenient set of methods allowing one to specify the type of read data and avoid tedious byte

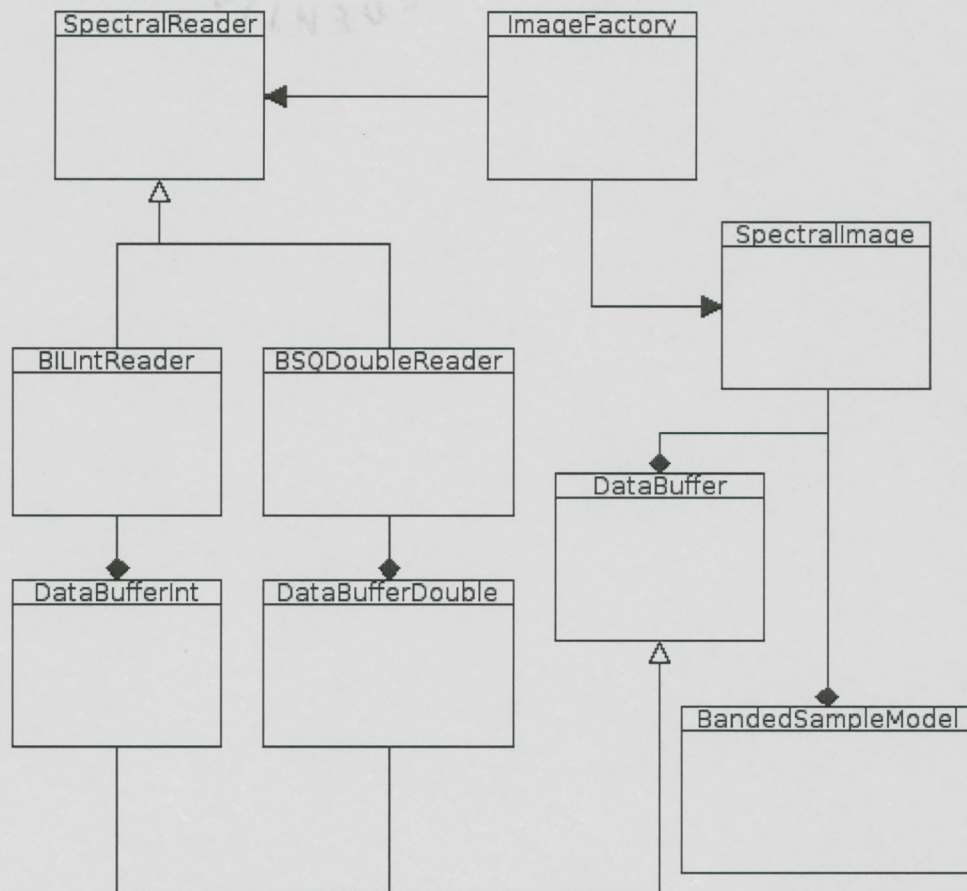


Figure 5.7: Partial Class Diagram of the I/O Framework

shifting, and type conversions and concentrate on the program logic.

Unfortunately most the methods of `RandomAccessFile` assume big endian byte order in the read files. This may sometimes coause problems as binary files generated on the Intel x86 platform are usually stored in the little endian notation. To counter this shortcoming we have developed the `maciak.hyper.io.ByteFlipper` class. It offers a very simple solution to a complex problem. If a file is encoded in little endian notation, we simply run it through `ByteFlipper` and convert it to big endian for further processing.

This is accomplished by reading in the file one word (4 bytes) at a time. Each word represented as a byte array of length 4 then reversed:

```
byte[] reversedWord = new byte[word.length];

for(int i=0, j=word.length-1; i<word.length; i++, j--)
    reversedWord[i] = a[j];
```

Each word is then written into a file, creating an exact copy of the original, only with opposite endian encoding. The conversion can be carried out both ways. For example, if necessary an output file written by the Java application can be converted back to little endian for further processing using Intel based tools using the exact same notation and method calls.

5.5 Command Line User Interface

In addition to a graphical user interface, we also wanted to have a command line mode of operation. Not having to load a GUI interface can be an advantage. Drawing windows on the screen, and capturing user events takes up memory and resources which could be better used elsewhere when processing large data sets. In addition it makes the code more portable. Some Unix and Linux SMP workstations may not have X server installed, or do not allow X forwarding. Therefore only way to run the processing algorithms on a headless remote machine like that is to have a command line mode.

Both the GUI and the CLI interfaces can be accessed by running the main method of the `maciak.hyper.Main` class. If the class is run without any command line parameters then the graphical mode is invoked. If on the other hand it is run with the

-c argument, it will then launch in the CLI mode. The full command line usage for this mode is as follows:

```
java -Xmx1024mb maciak.hyper.Main -c -t=threads -i=iterations -o=outbands
    -m=mode -s=seed -l=logging -k=stop-condition
```

The two first arguments are mandatory, while the last two are optional. The explanation of arguments can be found in Table 5.5.

Argument	Explanation	Default Value	Accepted Values
-t	number of threads to be used	1	any int
-i	maximum number of iterations	1000	any int
-o	desired number of output bands	same as orig	any int
-l	enable logging to a file	no	yes or no
-s	seed for random number generator	none	any int
-k	stop condition (kill value)	0.0001	any float < 1
-m	nmf algorithm to be used	standard	standard sprojected pprojected stochastic

Table 5.1: CLI Arguments

An example of usage could be:

```
java -Xmx1024mb maciak.hyper.Main -c -t=3 -i=2000 -o=20
    -m=pprojected -s=1234567890 -l=yes
```

or for the pre-compiled JAR file:

```
java -Xmx1024mb -jar hyp.jar -c -t=3 -i=2000 -o=20
    -m=pprojected -s=1234567890 -l=yes
```

The above commands would run the Parallel Projected Gradient algorithm for 1000 iterations, or until it convergers (whichever comes first) using 3 threads and outputting

an abundance array image with 20 bands. If the band specification is omitted then the output will have the same number of bands as the original image. If the algorithm is omitted, default NMF algorithm will be used.

In the CLI mode all the messages are written into standard output. When the application is started in the GUI mode however, the output is redirected to a log file `.hyperj_log` located in the same directory as `.hyperj`.

Chapter 6

Conclusions and Future Work

In this paper we set out to research investigate two new approaches to solving the Nonnegative Matrix Factorization problem when employed for feature extraction in hyperspectral imagery. Previous work partially reviewed in this document has shown that NMF is a viable, and effective tool in that research area. We devised two parallel algorithms based on known and tested NMF implementations with the aim of improving their overall performance when run on a multi processor machine. We then implemented these algorithms using Java, and tested them on a 4 CPU Solaris workstation

In both cases we were successful. Upon conducting a series of tests we concluded that our parallel implementations clearly outperform their sequential predecessors. In the case of standard NMF, we found that our algorithm is very flexible and scalable. Parallel Projected Gradient algorithm on the other hand could only be improved to some extent. Our implementation was shown not to be as scalable as the standard P-NMF one. However it still offers a notable improvement in performance.

Because of hardware limitations and time constraints we were unable to test our algorithms on machines with more than 4 CPU's. In our findings we made theoretical

assumptions about their scalability based on the collected data. In the future, we would like to conduct some more thorough testing on a SMP machine with 6-8 CPU's. This would help us to verify the claims about potential increase in speedup for our PPG-NMF implementation.

We would also like to investigate alternative methods of parallelization that could have been used for the PPG-NMF algorithm. The fact that our current implementation does not seem to scale with the number of available processors warrants further research into this subject. Developing a more scalable implementation would be highly desirable, because PPG-NMF is both faster and more accurate than the P-NMF.

A final logical continuation of this work would be extending P-NMF into a distributed cluster based implementation. This could be done using Java PRC API, or custom third party clustering tools outlined in this paper. Since we already have the parallelization framework, we would only have to develop and model efficient mechanism to share data between the parallel nodes without introducing too much communication overhead.

While working on this paper we have also developed an extensible, java based hyperspectral imaging framework, and a Graphical User Interface for our factorization tools. This framework is by no means complete, and could be extended to support new data types, or even feature extraction algorithms. It could be worthwhile to implement both sequential and parallel versions of other popular algorithms such as PCA, ICA and more.

Bibliography

- [1] D. Lee and H. Seung, "Learning the parts of objects by non-negative matrix factorization," *Nature*, vol. 401, pp. 788–791, 1999.
- [2] C.-J. Lin, "Projected gradient methods for non-negative matrix factorization," *Neural Computation*, 2007, to appear. [Online]. Available: <http://www.csie.ntu.edu.tw/~cjlin/papers/pgradnmf.pdf>
- [3] J. B. Campbell, *Introduction to Remote Sensing*. Guilford Press, 2002.
- [4] A. P. Cracknell and L. Hayes, *Introduction to Remote Sensing*. Taylor and Francis, 1991.
- [5] S. A. Robila, "Using spectral distances for speedup in hyperspectral image processing," *International Journal of Remote Sensing*, vol. 26, pp. 5629–5650, 2005.
- [6] A. Plaza, D. Valencia, J. Plaza, and C.-I. Chang, "Parallel implementation of endmember extraction algorithms for hyperspectral data," *Geoscience and Remote Sensing, IEEE Transactions on*, vol. 42, pp. 650–663, March 2004.
- [7] C. I. Chang, *Hyperspectral Imaging: Techniques for Spectral Detection and Classification*. Springer, 2006.
- [8] N. M. Short. (2006, March) Remote sensing tutorial. [Online]. Available: <http://rst.gsfc.nasa.gov/>

- [9] S. A. Robila and L. G. Maciak, "A parallel unmixing algorithm for hyperspectral images," D. P. Casasent, E. L. Hall, and J. Ronning, Eds. SPIE.
- [10] S. A. Robila and L. G. Maciak, "New approaches for feature extraction in hyperspectral imagery." IEEE LISAT, 2006.
- [11] N. Keshawa and J. Mustard, "Spectral unmixing," *IEEE Signal Processing Magazine*, pp. 47–57, 2002.
- [12] D. Landgrebe, "On information extraction principles for hyperspectral data," School of Electrical and Computer Engineering, Purdue University, 1997. [Online]. Available: <http://dynamo.ecn.purdue.edu/landgreb/whitepaper.pdf>
- [13] D. Landgrebe, "Multispectral data analysis: A signal theory perspective," School of Electrical and Computer Engineering, Purdue University, April 1998. [Online]. Available: http://cobweb.ecn.purdue.edu/beihl/MultiSpec/Signal_Theory.pdf
- [14] "Envi reference guide," ITT Visual Solutions, July 2006. [Online]. Available: <http://www.itvis/envi>
- [15] S. A. Robila, "Advanced image processing techniques for remotely sensed hyperspectral data: Independent component analysis," P. K. Varshney and M. K. Arora, Eds. Springer, 2004, pp. 109–132.
- [16] V. P. Pauca, J. Piper, and R. J. Plemmons, "Nonnegative matrix factorization for spectral data analysis," *Linear Algebra and its Applications*, vol. 416, no. 1, pp. 29–47, July 2006.
- [17] M. Chu and R. Plemmons, "Nonnegative matrix factorization and applications," *Bulletin of the International Linear Algebra Society*, vol. 34, pp. 2–7, 2005.
- [18] D. Lee and H. Seung, "Algorithms for non-negative matrix factorization," *Advances in Neural Processing*, 2000.

- [19] M. Chu, F. Diele, R. Plemmons, and S. Ragni, "Theory, numerical methods and applications of the nonnegative matrix factorization," *SIAM Journal on Matrix Analysis and Application*, 2004.
- [20] D. P. Bertsekas, *Nonlinear Programming*, 2nd ed. Belmont, MA: Athena Scientific, 1999.
- [21] S. Ingram, "An improved projected gradient method for nonnegative matrix factorization." [Online]. Available: http://aux.planetmath.org/files/papers/332/cs542_final.pdf
- [22] A. Cichocki and R. Zdunek, "Multilayer nonnegative matrix factorization," *Electronics Letters*, vol. 42, pp. 947–948, 2006. [Online]. Available: http://www.bsp.brain.riken.jp/publications/2006/CichZd_EL06-rev-www-final.pdf
- [23] A. Plaza, P. Martinez, R. Perez, and J. Plaza, "A quantitative and comparative analysis of endmember extraction algorithms from hyperspectral data," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 42, March 2004.
- [24] I. T. Jolliffe, *Principal Component Analysis*. Springer, 2002.
- [25] S. A. Robila and P. K. Varshney, "Advanced image processing techniques for remotely sensed hyperspectral data: Feature extraction from hyperspectral data using ica," P. K. Varshney and M. K. Arora, Eds. Springer, 2004, pp. 199–216.
- [26] J. Dongarra, I. Foster, G. C. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, *The Sourcebook of Parallel Computing*. Morgan Kaufmann, 2002.
- [27] A. Ferrari, "Jpvm: Network parallel computing in java," *Concurrency: Practice and Experience*, vol. 10, pp. 985–992, 1998.

- [28] A. J. Plaza, "Hyperspectral data exploitation, theory and applications: Morphological hyperspectral image classification - a parallel processing perspective," C.-I. Chang, Ed. Wiley, 2007, pp. 353-378.
- [29] A. Plaza, D. Valencia, J. Plaza, and P. Martinez, "Commodity cluster-based parallel processing of hyperspectral imagery," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 345-358, March 2006.
- [30] S. H. Roosta, *Parallel Processing and Parallel Algorithms*. Springer, 1999.
- [31] T. Achalkul and S. Taylor, "A distributed spectral screening pct algorithm," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 373-384, March 2003.
- [32] "Java sdk 1.4," Sun Microsystems, 2006. [Online]. Available: <http://java.sun.com>
- [33] B. Lewis and D. J. Berg, *Multithreaded Programming with Java Technology*. Prentice Hall PTR, 1999.
- [34] S. Oaks and H. Wong, *Java Threads*. O'Reilly, 1999.
- [35] S. Microsystems, "The java tutorials: Concurrency," 1995-2006. [Online]. Available: <http://java.sun.com/docs/books/tutorial/essential/concurrency/>
- [36] "Eclipse java ide," IBM. [Online]. Available: <http://eclipse.org>
- [37] S. Holzner, *Eclipse: Programming Java Applications*. O'Reilly, 2004.
- [38] "Hyperspectral digital imagery collection experiment documentation," August 1995.
- [39] *Programmer's Guide to the Java 2D API: Enhanced Graphics and Imaging for Java*. Sun Microsystems, Inc., April 2004.

- [40] R. Eckstein, J. Elliott, B. Cole, D. Wood, and M. Loy, *Java Swing*. O'Reilly, 2002.
- [41] "The standard widget toolkit," IBM. [Online]. Available: <http://eclipse.org/swt>
- [42] I. Kriznar, "Swt vs. swing performance comparison," Cosylab D.O.O, 2005. [Online]. Available: http://cosylib.cosylab.com/pub/CSS/DOC-SWT_Vs._Swing_Performance_Comparisson.pdf

Appendix A

Source Code

This appendix contains additional information regarding the source code of the applications used in this work. The code is organized in the following file structure:

```
thesis                                [root directory]
|
+-----doc                            [javadoc documentation]
|
+-----maciak                          [root project directory]
  |
  +-----msc                            [assorted misc classes]
  |
  +-----hyper                          [hyperspectral processing classes]
    |
    +-----gui                           [gui related classes]
    |
    +-----img                           [image processing classes]
    |
    +-----io                            [i/o and encoding classes]
```


The maciak directory is the root level package which contains all the thesis related code. Inside one can find the msc package containing assorted classes that deal with tasks such as type conversions or handling Java Properties. The hyper package contains the Main class used to launch the GUI interface or command line client and three sub packages: img, gui and io.

The img package contains all the classes that deal with handling hyperspectral data, as well as the actual factorization code.

The gui package contains all the files related to displaying the graphical user interface and handling user interaction.

Finally, the io package contains classes dealing with input and output.

A.1 Documentation

The doc directory contains javadoc generated HTML documentation describing the publicly visible API's of all the classes. Also see the Chapter 5 and this appendix for more information.

A.2 Compiling the Code

There are no special requirements for compiling the code. Sun's version of Java 1.4 or better is required to compile and run the samples.

A.3 Running The Code

See Section 5.3.4 for details on how to start the GUI interface, and Section 5.5 for how to run the application from the command line. Table 5.5 show all the available command line switches.

A.4 Extending the Toolkit

At the moment there is no plug in support but given more time it would be built into the system. When adding new features it will be necessary to recompile certain files.

A.4.1 Adding new Reader or Writer Class

To add a new Reader/Writer class for a new file type, simply extend the base `maciak.hyper.io.SpectralReader` or `maciak.hyper.io.SpectralWriter` class. Once this is done, simply add a single line to `getReaderByType` or `getWriterByType` method in that base class and recompile it. The new functionality will then become available as long as correct data type parameters are specified in `.hyperj` file or via the GUI dialog.

A.4.2 Adding new Factorization Algorithm

To add a new factorization algorithm, extend the `maciak.hyper.img.Factorizer` class. You will need to edit the `maciak.hyper.Main` and potentially some of the classes in `maciak.hyper.gui` to add a new command line switch, and/or GUI button to launch the new algorithm.

A.4.3 Adding new DataBuffer

Not all data types commonly used in hyperspectral imaging are supported by the java.awt. It should be relatively easy to create new DataBuffer subclasses that would accommodate these non-standard types. See the Java API for implementation details.

A.5 Known Issues

There are some known issues with the code that we haven't worked out yet. The visualization feature is very slow. Displaying large images in Swing does not seem to work well, and some sort of partial loading, and caching algorithm is needed to resolve this. Alternatively it might be necessary to port the visualization features to a 3rd party widget toolkit.

In addition some of the buttons in the GUI mode do not work properly. Since our main testing machine had no X11 installation, we mainly concentrated on the CLI interface.