MRI Velocity Quantification Implementation and Evaluation of Elementary Functions for the Cell Broadband Engine

MRI Velocity Quantification Implementation and Evaluation of Elementary Functions for the Cell Broadband Engine

By
Wei Li, M.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M. A. Sc
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE (2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
MRI Velocity Quantification Implementation and
Evaluation of Elementary Functions for the Cell Broadband Engine

AUTHOR: Wei Li, M.Sc. (Northeastern University, CHINA)

SUPERVISOR: Dr. Christopher Kumar Anand

NUMBER OF PAGES: viii, 83

Abstract

Magnetic Resonance Imaging (MRI) velocity quantification is addressed in part I of this thesis. In simple MR imaging, data is collected and tissue densities are displayed as images. Moving tissue creates signals which appear as artifacts in the images. In velocity imaging, more data is collected and phase differences are used to quantify the velocity of tissue components. The problem is described and a novel formulation of a regularized, nonlinear inverse problem is proposed. Both Tikhonov and Total Variation Regularization are discussed. Results of numerical simulations show that significant noise reduction is possible.

The method is firstly verified with MATLAB. A number of experiments are carried out with different regularization parameters, different magnetic fields and different noise levels. The experiments show that the stronger the complex noise is, the stronger the magnetic field requires for estimating the velocity. The regularization parameter also plays an important role in the experiments. Given the noise level and with an appropriate value of regularization parameter, the estimated velocity converges to ideal velocity very quickly. A proof-of-concept implementation on the Cell BE processor is described, quantifying the performance potential of this platform.

The second part of this thesis concerns the evaluation of an elementary function library. Since CBE SPU is designed for compute intensive applications, the well developed Math functions can help developer program and save time to take care other details. Dr. Anand's research group in McMaster developed 28 math functions for CBE SPU. The test tools for accuracy and performance were developed on CBE. The functions were tuned while testing. The functions are either competitive or an addition to the existing SDK1.1 SPU math functions.

Acknowledgements

I would like to thank my supervisor, Dr. Christopher Anand, for his continuous encouragement and guidance during the period when I worked on this thesis. He spent much of his invaluable time with me discussing my research, reviewing the draft of this thesis. His solid knowledge and experience in MRI and optimization and his advanced programming skills as well as his working style and devotion to science and research, have inspired me during my study and will continually influence me in my future research and career.

I would like also to thank IBM Toronto lab, especially Robert Enenkel, Marin Litoiu, Roland Koo, Kelly Lyons, to offer me the CAS fellowship and supplemental position, which made me have the opportunity to work in IBM and enrich my thesis.

I am grateful to my classmates for their friendship and help as well as all staff members in the Department of Computer and Software Engineering, especially Dr. Wolfram Kahl and Dr. Spencer Smith, for their reviewing my thesis and valuable comments. The patience, help and full cooperation of the clerical staff in the general office of the Department are also acknowledged.

Finally, I must express my heartfelt gratitude to my family members. especially my parents and my husband, for their love and encouragement. Without their never-ending support, I would not have completed my study.

Notations

CBE:	Cell Broadband Engine
DMA:	Direct Memory Access
FP:	Floating Point
MFC:	Memory Flow Controller
MRI:	Magnetic Resonance Imaging
PPE:	PowerPC Processor Element
PPU:	PowerPC Processor Unit
SPE:	Synergistic Processing Element
SPU:	Synergistic Processor Unit
TV:	Total Variation

Contents

List of Figures

List of Tables

Chapter 1

Introduction

MRI is medical diagnosis equipment widely used in Hospitals and the Cell Broadband Engine (CBE) processor is a novel processor developed by Sony, IBM and Toshiba.

As the first part of this thesis, a computationally intensive problem in MRI (Velocity Quantification) is addressed. An iterative algorithm for MRI velocity Quantification is first proposed and an implementation of the algorithm is made on CBE. During the implementation, some math functions, e.g., sin, cos and exponent, are needed. As we known, a math library plays an important role for a software development for a computational application. Usually the math functions are released with information of performance and accuracy. However, the existing CBE SPU math library SPU in SDK 1.1 does not provide these information and some math functions does not appear in the library. Performance and accuracy testing on the new developed math functions for CBE SPU is the second part of this thesis. With the tested math functions, one can safely call the functions for software development.

The iterative algorithm for MRI velocity quantification, incorporated with total variation regularization is introduced in Chapter 3. The algorithm is firstly simulated in MATLAB. The algorithm is further implemented with a real processor, called CBE. To accommodate the CBE architecture, especially the DMA transfer limitation, an algorithm of solving linear equations is proposed. The results are consistent with the ones from the simulation with MATLAB. The CBE implementation is only a proof-of-concept for gathering benchmark information, and would require substantial rewriting to handle realistic problem sizes.

To have a basic understanding of MRI and CBE, the next two sections will give an overview for both.

1.1 Overview of MRI

Magnetic resonance imaging (MRI) is an imaging technique used primarily in medical settings to produce high quality images of the inside of the human body. MRI is based on the principles of nuclear magnetic resonance (NMR), a spectroscopic technique used by scientists to obtain microscopic chemical and physical information about molecules.

Atoms with an odd number of protons and/or odd number of neutrons possess a nuclear spin angular momentum, and therefore exhibit the MR phenomenon. Qualitatively, these nucleons can be visualized as spinning charged spheres that give rise to a small magnetic moment. We often refer to these MR-relevant nuclei as simply spins [3].

The nature of MR is based on the intersection of these spins with three types of magnetic fields:

- Main field
- Radiofrequency field
- Linear gradient fields

Nuclei are excited by a radiofrequency magnetic field and produce a time signal which is recorded and processed to form an image. Magnetic resonance started out as a tomographic imaging modality for producing NMR images of a slice through the human body. Each slice had a thickness (Thk). The slice is said to be composed of several volume elements or voxels. The volume of a voxel is approximately 3 mm^3. The magnetic resonance image is composed of several picture elements called pixels. The intensity of a pixel is proportional to the NMR signal intensity of the contents of the corresponding volume element or voxel of the object being imaged.

Magnetic resonance imaging is based on the absorption and emission of energy in the radio frequency range of the electromagnetic spectrum.

1.1.1 Image Artifacts

An image artifact is any feature which appears in an image but is not present in the original imaged object. An image artifact is sometimes the result of improper operation of the imager, and other times a consequence of natural processes or properties of the human body. Artifacts are typically classified as to their source. The following table summarizes a few of these.

Table 1.1: Typical MRI Artifacts

Artifact	**Common Cause**
RF Quadrature	Failure of the RF detection circuitry
B_0 Inhomogeneity	Metal object distorting the B_0 field
Gradient	Failure in a magnetic field gradient
RF Inhomogeneity	Failure of RF coil
Motion	Movement of the imaged object during the sequence
Flow	Movement of body fluids during the sequence
Chemical Shift	Resonance frequency / susceptibility differences between tissues
Partial Volume	Large voxel size
Wrap Around	Improperly chosen field of view

In this thesis, our focus is on velocity quantification. Motion artifacts are described in the next section.

1.1.2 Motion Artifacts

As the name implies, motion artifacts are caused by motion of the imaged object or a part of the imaged object during the imaging sequence. There are two major types:

- Motion due to flow, as found in blood vessels.
- Translational motion of tissue, such as the motion of the chest wall during breathing.

During acquisition of the data (flow or translational), the image will be blurred due to the motion above, possibly contaminated with other artifacts depending on the type of motion and image reconstruction used. Most motion artifacts found in MR images are due to a combination of flow and translational motion artifacts. The motion of the entire object during the imaging sequence generally results in a *blurring* of the entire image with ghost images in the phase encoding direction. Movement of a small portion of the imaged object results in a blurring of that small portion of the object across the image [1].

In practice, eliminating these artifacts is different for the two sources of motion.

- For flow (or motion) through a gradient, by appropriately adjusting the gradient amplitudes and timings along the direction of motion, the phase can be returned to zero at the echo so that it appears as if the imaged spins had been excited just prior to being imaged.

- Translational motion between RF pulses must be accounted for by freezing the motion, or by keeping track of the motion and either correcting for it or only collecting data when the tissue-of-interest is at one spatial location, effectively freezing it.

A disadvantage of these techniques is that the adjustment is determined by the heart rate or respiration rate which varies with people. We will propose an algorithm to estimate the velocity of motion. Then the contaminated image can be reconstructed, see [17]-[25].

1.2 Overview of Cell Broadband Engine (CBE)

A CBE is a 9 core processor. One of these cores is a PowerPC and acts as a controller. The remaining 8 cores are called SPEs (Synergistic Processing Element) and these are very high performance vector processors. Each SPE contains its own block of high speed RAM and is capable of 32 GigaFlops (32 bit). The SPEs are independent processors and can act alone or can be set up to process a stream of data with different SPEs working on different stages. This ability to act as a "stream processor" gives

access to the full processing power of a CBE which is claimed to be more than 10 times higher than even the fastest desktop processor [5, 6, 7].

Although the CBE Broadband Engine is initially intended for application in game consoles and media-rich consumer-electronics devices such as high-definition televisions, the architecture and the CBE Broadband Engine implementation have been designed to enable fundamental advances in processor performance, see [5]. It is hoped that it can be applied to a wide variety of applications. We are going to implement the proposed algorithm with CBE, to measure its performance.

1.2.1 CBE Architecture

CBE is an architecture for high performance distributed computing. It is comprised of hardware and software Cells. Software Cells consist of data and programs, these are sent out to the hardware Cells where they are computed, the results are then returned. An individual hardware CBE consists of the following elements [7]:

- 1 Power Processor Element (PPE).
- 8 Synergistic Processor Elements (SPEs).
- Element Interconnect Bus (EIB).
- Direct Memory Access Controller (DMAC).
- 2 Rambus XDR memory controllers.
- Rambus FlexIO (Input / Output) interface.

The architecture is visualized in Figure 1.1, see [6].

The Power Processor Element (PPE)

The PPE is a conventional microprocessor core which sets up tasks for the SPEs to do. In a CBE based system the PPE will run the operating system and most of the applications but compute intensive parts of the OS and applications will be offloaded to the SPEs [7]. The features of PPE include

- a 64 bit, "Power Architecture" processor with 512K cache

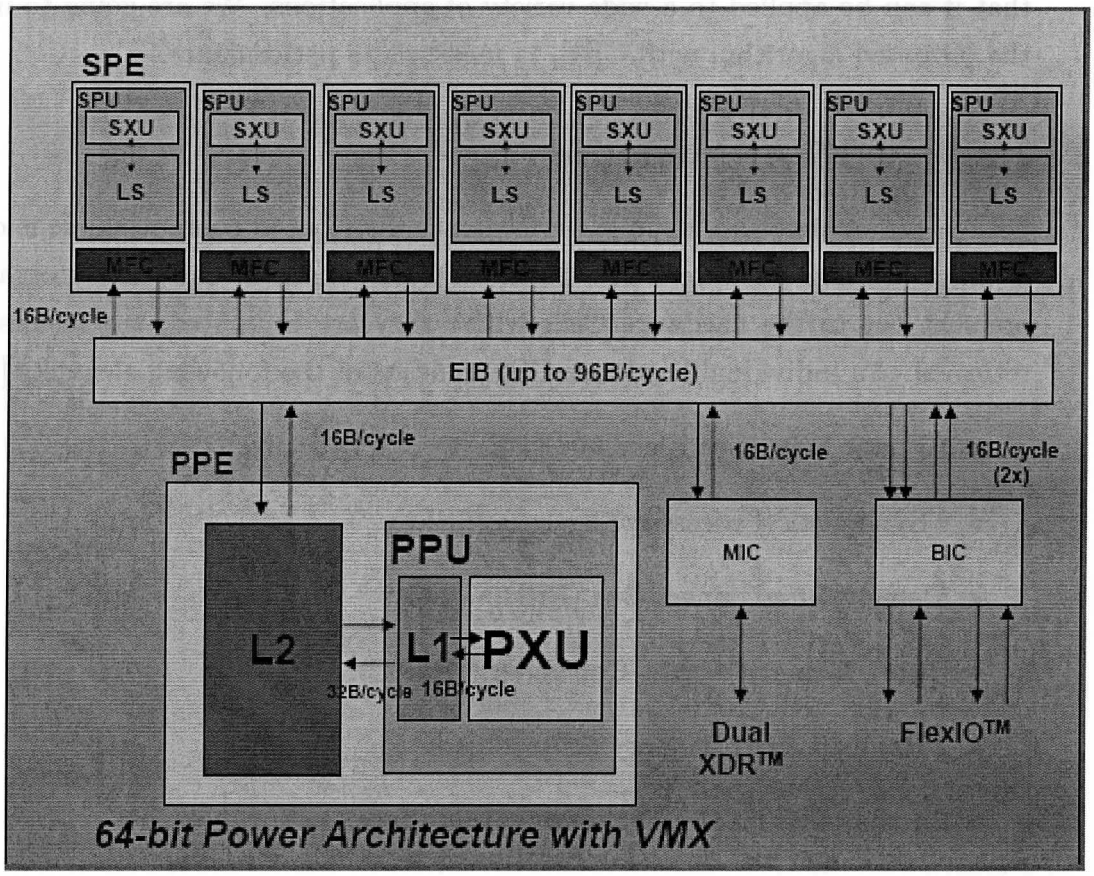

Figure 1.1: CBE Processor Architecture

- 2-Way hardware multithreaded
- Using the PowerPC instruction set and supporting for the VMX vector instructions.
- Coherent load/store
- Realtime controls

Synergistic Processor Element (SPE)

An SPE is a self contained vector processor which acts as an independent processor. The features of SPE include

- Simple RISC User Mode Architecture–Dual issue VMX-like, Graphics SP-Float, IEEE DP-Float
- Resources: 128 128 bits registers, 256KB Local Store
- Dedicated DMA engine: Up to 16KB outstanding requests
- Providing the computational performance

1.2.2 Software Development on CBE

It is said that CBE is difficult to develop for because it requires a different way of thinking. However, if one is involved in a real application development with CBE, a similar developing steps can be used. For most developers, compilers, operating systems, libraries and middleware will take care of a lot of the fine detail [7].

The primary language for developing on the CBE is expected to be C with normal thread synchronisation techniques used for controlling the execution on the different cores. C++ is also supported to a degree and other languages are also in development.

The PPE just puts jobs into the job queue. The SPE loads up the data via DMA transfer and gets computing. The SPEs are dynamically assigned, the developer does not need to worry about which SPE is doing what.

The development environment can be show in Figure 1.2, see [6]

CBE Development Tools

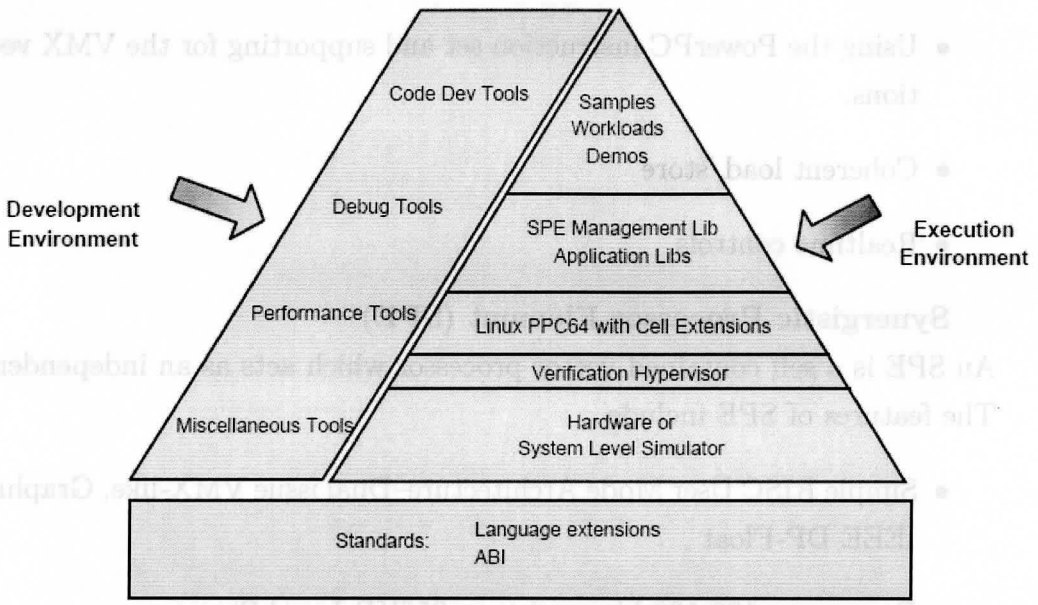

Figure 1.2: CBE Development Environment

- C/C++ compiler
  - GNU based C/C++ compiler for both PPE & SPE
  - IBM XLC C/C++ optimizing compiler for both PPE & SPE (ppuxlc & spuxlc)
- Debugger: GNU gdb for both PPE & SPE
- SPE performance tools:
  - Static analysis (SPU_timing)
  - Dynamic analysis capabilities of the full system simulator
  - HW dynamic analysis
- SDK Libraries for PPE and SPE

Under the development environment, once the software is compiled, the developer can run it either on hardware or system level simulator. Running on the simulator

can help developer to analyze the code and improve the performance.

CBE programming issues [6]

Programming on CBE is not an easy job. The following tips can help developers to develop code on PPE and SPE.

- Algorithm complexity study
- Data layout/locality and Data flow analysis
- Experimental partitioning and mapping of the algorithm
- program structure to the architecture
- Develop PPE Control, PPE Scalar code
- Develop PPE Control, partitioned SPE scalar code– communication, synchronization, latency handling
- Transform SPE scalar code to SPE SIMD code
- Re-balance the computation / data movement
- Other optimization considerations–PPE SIMD, system bottle-neck, load balance

Based on the above, programming on CBE will be presented in Chapters 4 and 6.

1.3 Outline of the Thesis

This thesis is divided into two parts. Part 1, consisting of Chapter 2, Chapter 3 and Chapter 4, investigates MRI velocity quantification and implementation on CBE. Part 2, composing of Chapter 5 and Chapter 6, deals with floating-point math function test with single precision. The chapters are organized in the following way:

Chapter 2 reviews the mathematical foundation for developing the algorithm

for MRI velocity quantification.

Chapter 3 focuses on the iterative algorithm development for MRI velocity quantification.

Chapter 4 touches on the implementation of the algorithm on CBE.

Chapter 5 provides the introduction on floating-point number with single format.

Chapter 6 deals with the accuracy and performance test on SPU SIMD math functions.

Chapter 7 contains conclusions and future work.

Part I

MRI Velocity Quantification Algorithm Development and Implementation

Chapter 2

Mathematical Foundation

2.1 Tikhonov Regularization

In mathematics, inverse problems are often ill-posed. To solve these problems numerically, we have to introduce some extra information about the solution, such as an assumption on the smoothness or a bound on the norm. This information is called a *priori* knowledge, a knowledge which is known beforehand. The choice of which type of information is imposed on the solution has a high impact on the computed solutions. One most used method of regularization is to suppress the errors on the computed solution with a priori knowledge, which forms a new problem, called the Tikhonov regularization method.

2.1.1 Definition

Let us consider a linear system described by the following equation:

$$Ax = b + noise \tag{2.1}$$

where A is an mxn matrix, x is a column vector with length of n and b is a column vector with length of m. This is an overdetermined problem. The objective is to seek x by minimizing $T(x)$. The problem can be replaced by

$$\min_x T(x) \tag{2.2}$$

where

$$T(x) = ||Ax - b||_2^2 + \lambda P(x) \tag{2.3}$$

where T(x) is the objective function. $||Ax - b||_2^2$ is referred to the fitting term. $P(x)$ is called the penalty term and λ is the regularization parameter.

The formulation described in Equations 2.2 and 2.3 is referred to the Tikhonov regularization.

2.1.2 Penalty Function $P(x)$

The penalty term is sometimes called a discrete smoothing norm. That means that high frequencies in the solution are penalized and meanwhile low frequencies are unregularized (i.e., allowed go through). There are several ways to choose the penalty function, $P(x)$, see Reference [14].

- Case 1: In the standard form of Tikhonov regularization, the penalty term $P(x) = ||x||_2^2$.
- Case 2: If a priori knowledge about the solution is possible, the penalty term can be chosen as $P(x) = ||Dx||_2^2$, where D can be a weighting matrix or a discrete derivative operator.

Note that case 1 is a special case of case 2 with $D = I$. In both cases, the 2-norm is used and very smooth solutions result.

2.1.3 Regularization Parameter

The regularization parameter λ is a weight factor, which measures how important the penalty term is. we are going to show the effect of the regularization parameter in the following example.

Let us consider the problem described in Equations 2.2 and 2.3 and choose the penalty term in Case 1 mentioned as above, i.e., $P(x) = ||x||_2^2$. An explicit solution, denoted by $\hat{x}$, is given by:

$$\hat{x} = (A^T A + \lambda I)^{-1} A^T b \tag{2.4}$$

where I is the nxn identity matrix.

It is easily observed that If $\lambda = 0$, there is no penalty at all, which reduces to the least squares solution of an overdetermined problem ($m > n$). This is the simplest form of the Tikhonov regularization method. If $\lambda = \infty$, the fitting term does not affect the solution, and only the penalty function influences the solution. The solution is just a flat line. In these two extreme cases, the solution does not make much sense in practice. There is a trade-off to choose an optimal λ for a particular application. Finding an optimal regularization parameter itself is a very difficult topic, which can be found in [9]-[11]. We will not work on this topic theoretically in this project. Instead we will try to find a good regularization parameter from experiments and see how the parameter affects the solution in the next chapter.

Tikhonov regularization method provides a way to formulate the problem, but it does not tell how to solve the problem. What we know is that the solution can be found by minimizing the summation of the fitting term and the penalty term, which obviously falls into the optimization problem. A number of techniques have been published to solve the problem. For example,

- optimization-based
- filtering-based
- iteration-based

Selection of methods depends on a particular application. If the application is complicated, iterative method is a better choice by considering the complexity and memory use [12].

2.2 Total Variation (TV)

Total variation is a regularization technique that considers that the data is blocky and discontinuous. Most of the regularization methods assume the data to be smooth and continuous, but total variation discards this assumption. It measures the discontinuities in the data set [13].

Tikhonov regularization as we described in Section 2.1 uses norm ℓ_2, which results very smooth solution. The smooth solution is desirable in many applications while others require discontinuity or steep gradient to be computed. One approach is to replace norm ℓ_2 in Tikhonov regularization with the norm ℓ_1, i.e., the 1-norm of the first spatial derivation of the solution. This is called the total variation (TV) regularization. This method will help to obtain the discontinuities or steep gradients in the restored image [13]. The total variation can be expressed as

$$T(x) = ||Ax - b||_2^2 + \lambda \cdot TV(x) \tag{2.5}$$

where

$$TV(x) = \int_{\Omega} |\nabla x(t)| dt \tag{2.6}$$

where Ω is the domain, x is a function of one variable t and

$$|\nabla x(t)| = ((\frac{\partial x}{\partial t})^2)^{1/2} \tag{2.7}$$

From the above definition, it is found that TV is an integration of the length of all the gradients in any point in the domain Ω. As the gradient in a point is a measure of the variation of the function in this point, an integration over the entire domain must result in the total variation hence the name [12].

The one dimensional total variation functional can be approximated by using a discrete forward approximation

$$TV(x) \approx \sum_{i=1}^{n-1} \frac{|x_{i-1} - x_i|}{t_i} = \frac{1}{t} \sum_{i=1}^{n-1} |x_{i-1} - x_i| \tag{2.8}$$

where the equality assumes that the sampling points are equidistant, i.e., $t_i = t$ for $\forall i$.

2.3 Optimization

In Section 2.2, minimizing Equation 2.5 is obviously an optimization problem. There are various methods to obtain this minimization. We only describe the part needed for solving our problem.

The definitions and theorem of optimization can refer to [26, 27].

Let us approximate the function T with Taylor expansion with an arbitrary precision around an expansion point x_0.

$$T(x_0 + h) = T(x_0) + h^T T'(x_0) + \frac{1}{2} h^T T''(x_0) h + O(\| h \|^3) \tag{2.9}$$

where the term $O(\| h \|^3)$ represents all terms higher than second order. First or second order Taylor expansions are often used to model arbitrary continuous and differentiable functions of higher order. As long as the solution is very close to the expansion point x_0, i.e., h is small, then the expansion will be a good approximation to the function. In this case, we have

$$T'(x_0) = -\frac{1}{2} T''(x_0) h \tag{2.10}$$

Equation 2.10 forms a set of linear equations. The first derivatives (gradient) and the second derivatives of T can be computed as below.

The gradient of the function $T : R^n \rightarrow R$ is defined as

$$\nabla T(x) = T'(x) = \begin{bmatrix} \frac{\partial T}{\partial x_1} \\ \vdots \\ \frac{\partial T}{\partial x_n} \end{bmatrix} \tag{2.11}$$

The Hessian matrix of the same function T are defined by

$$T''_{ij}(x) = \frac{\partial^2 T}{\partial x_i \partial x_j} \tag{2.12}$$

Note: the Hessian matrix is symmetric and positive definite.

In Chapter 3, we are going to use the above fundamental to derive our algorithm.

2.4 Summary

This chapter introduced the mathematical foundation prepared for the next chapter. First, Tikhonov regularization method is described with different penalty functions.

These penalty functions all use the 2-norm and thus produce overly smooth solutions. Second, total variation is introduced to overcome the problems (i.e., too smooth and discontinous in the solution) by using the 1-norm. Third, one of the methods, optimization technique, is taken into account to solve the TV regularization.

Chapter 3

Algorithm Development For MRI Velocity Quantification

3.1 Physical Model

In Chapter 1, MRI and the artifacts due to the motion were briefly described. In this chapter, we are going to focus on study of velocity quantification using iterative method. In order to derive the algorithm, the mathematical model of velocity will be presented first.

Let us consider plug flow in a cylindrical vessel where all spins flow at the same velocity. The term "plug flow" implies that all of the spins within a given voxel have the same velocity. Further, we assume that the velocity is constant in time for every phase encoding step.

Suppose the flow along the read direction (the method could be applied to the other imaging directions). Spins moving with constant velocity in the read direction will develop a phase

$$\phi_{v\pm} \equiv \phi_{v_x}(T_E) = \mp\gamma v_x \tau^2 = \mp G v_x \tau^2 \tag{3.1}$$

By setting up the gradient appropriately, the velocity dependent phase generated by the bipolar pulses is not affected by the subsequent compensation gradients. Meanwhile, the fact of the specific phase corresponding to velocity in the reconstructed images can be employed to measure blood flow and/or velocity.

The model based on such assumptions can be described as [2]

$$I_G(x,y) = e^{-iG\cdot v_x(x,y)}\rho(x,y) + noise \tag{3.2}$$

where $v_x(x,y)$ is the two dimensional velocity with size $M \times M$ and we assume that it changes along readout direction only, i.e., along x axe in this thesis. $\rho(x,y)$ is the tissue density with two dimensional with size $M \times M$. G is the gradient magnetic field strength and it is a design parameter. $I_G(x,y)$ is the two dimensional image of size $M \times M$ with the gradient field. The noise refers to the environmental noise.

The model implies that velocity changes the phase of the image only and does not change amplitude of the image. That means the amplitudes of measured images are same with the original density.

3.2 Total Variation (TV) Regularization

Given the physical model (3.2) and measured 2 dimensional (2-d) images with different G, one can solve the velocity by minimizing the noise under the norm 2, see [2] . That is

$$\min \sum_{x,y} ||I_G(x,y) - e^{-iG\cdot v_x(x,y)}\rho(x,y)||_2 \tag{3.3}$$

This is nonlinear optimization problem. By using the Newton's method, the numerical solution of velocity can be approximately estimated. We also call the formula (3.3) as a fitting term. The problem arises, however, when the measured images have values of zeros, resulting in an ill-posed solution. In addition, even though the measured images are all of non-zero values, the high frequent components, e.g. due to the noise, will produce a non reasonable solution of velocity. To overcome these problems, some measure of the solution as a penalty function is taken into account. As we discussed in Chapter 2, we choose total variation (TV) of the velocity as the measure of solution [2].

By combing the fitting term and the TV of the velocity, the optimization problem can be formulated as

$$\min\{\sum_{x,y} ||I_G(x,y) - e^{-iG\cdot v_x(x,y)}\rho(x,y)||_2 + \lambda \cdot TV(v)\} \tag{3.4}$$

where λ is the regularization parameter. $TV(v)$ is often taken as gradient magnitude in the image processing and will be discussed in the next section.
For convenience, let

$$T(v) = T_1(v) + T_2(v), \tag{3.5}$$

where

$$T_1(v) = \sum_{x,y} ||I_G(x,y) - e^{-iG\cdot v_x(x,y)}\rho(x,y)||_2 \tag{3.6}$$

$$T_2(v) = \lambda \cdot TV(v) \tag{3.7}$$

The optimization problem is to find a minimizer v_x such that the function $T(v)$ achieves the minimum value, i.e.,

$$\min_{v_x} T(v) \tag{3.8}$$

3.3 Algorithm Development

For the optimization problem specified in (3.8), we will adopt the Newton's method. The Newton method has been widely used for solving unconstrained linear optimization problems. In principle, the key of the Newton's method (also called Newton-Raphson) is to construct the Jacobian vector and Hessian matrix with reference to Section 2.3. Because of the nonlinearity in both the fitting term and TV term, it is not easy to calculate the Hessian, especially for the TV term. Let us examine the Jacobian and and Hessian of the two terms, respectively.

3.3.1 The Fitting Term ($T_1(v)$)

As we know, the Jacobain vector consists of the first derivatives of T. After discretization, the fitting term T_1 becomes

$$T_1 = \sum_{i,j}[(I_{R_{i,j}} - \rho\cos(Gv_{x_{i,j}}))^2 + (I_{I_{i,j}} + \rho\sin(Gv_{x_{i,j}}))^2] \tag{3.9}$$

where $I = I_R + iI_I$.

Note: For simplicity, I_G is replaced by I.

The gradient of the function T_1 w.p.t $v_{k,m}$ is derived as

$$\frac{\partial T_1}{\partial v_{x_{k,m}}} = 2\rho G[I_{R_{k,m}}\sin(G \cdot v_{x_{k,m}}) + I_{I_{k,m}}\cos(G \cdot v_{x_{k,m}})] \tag{3.10}$$

Let $N = M \times M$. The Jacobian vector $(N \times 1)$ of T_1 can be formed as

$$T_{1_{Jac}} = \begin{bmatrix} \frac{\partial T_1}{\partial v_{x_{1,1}}} & \frac{\partial T_1}{\partial v_{x_{1,2}}} & \cdots & \frac{\partial T_1}{\partial v_{x_{1,M}}} & \cdots & \cdots & \frac{\partial T_1}{\partial v_{x_{M,1}}} & \frac{\partial T_1}{\partial v_{x_{M,2}}} & \cdots & \frac{\partial T_1}{\partial v_{x_{M,M}}} \end{bmatrix}^T \tag{3.11}$$

The second derivatives of the function T_1 with respect to $v_{x_{k,m}}$ is

$$\frac{\partial^2 T_1}{\partial v_{x_{p,q}} \partial v_{x_{k,m}}} = \begin{cases} 2\rho G^2[I_{R_{p,q}}\sin(G \cdot v_{x_{p,q}}) + I_{I_{p,q}}\cos(G \cdot v_{x_{p,q}})] & \text{if } p = k, q = m \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

The Hessian matrix $(N \times N)$ of T_1 can be formed as

$$T_{1_{Hess}} = \begin{bmatrix} \frac{\partial^2 T_1}{\partial v_{x_{1,1}}^2} & 0 & \cdots & 0 & 0 \\ 0 & \frac{\partial^2 T_1}{\partial v_{x_{1,2}}^2} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{\partial^2 T_1}{\partial v_{x_{M,M-1}}^2} & 0 \\ 0 & 0 & 0 & \cdots & \frac{\partial^2 T_1}{\partial v_{x_{M,M}}^2} \end{bmatrix} \tag{3.13}$$

Obviously, $T_{1_{Hess}}$ is a diagonal matrix.

3.3.2 The TV Term ($T_2(v)$)

As we know, the TV functional is nonlinear and gives rise to troubles for solving its derivatives. As introduced in Section 2.2, the discrete form of T_2 can be expressed as

$$T_2 = \sum_{i,j} |v_{x_{i+1,j}} - v_{x_{i,j}}| \tag{3.14}$$

According to Reference [2], T_2 can be approximated with a quadratic function in the following way

$$T_2 \simeq \sum_{i,j} \frac{1}{2|\tilde{v_{x_{i+1,j}}} - \tilde{v_{x_{i,j}}}| + \epsilon}(v_{x_{i+1,j}} - v_{x_{i,j}})^2 \tag{3.15}$$

The first partial derivative is shown as below:

$$\begin{aligned} \frac{\partial T_2}{\partial v_{x_{k,m}}} &= \left(\frac{2}{2|\tilde{v}_{x_{k,m}} - \tilde{v}_{x_{k-1,m}}| + \epsilon} + \frac{2}{2|\tilde{v}_{x_{k+1,m}} - \tilde{v}_{x_{k,m}}| + \epsilon}\right)v_{x_{k,m}} \\ &- \frac{2}{2|\tilde{v}_{x_{k,m}} - \tilde{v}_{x_{k-1,m}}| + \epsilon}v_{x_{k-1,m}} - \frac{2}{2|\tilde{v}_{x_{k+1,m}} - \tilde{v}_{x_{k,m}}| + \epsilon}v_{x_{k+1,m}} \end{aligned} \tag{3.16}$$

where ϵ is a small positive number to avoid the dominator of zero.

Similar to the fitting term, the Jacobian vector ($N \times 1$) of T_2 can be formed as

$$T_{2_{Jac}} = \left[\frac{\partial T_2}{\partial v_{x_{1,1}}} \quad \frac{\partial T_2}{\partial v_{x_{1,2}}} \quad \cdots \quad \frac{\partial T_2}{\partial v_{x_{1,M}}} \quad \cdots \quad \cdots \quad \frac{\partial T_2}{\partial v_{x_{M,1}}} \quad \frac{\partial T_2}{\partial v_{x_{M,2}}} \quad \cdots \quad \frac{\partial T_2}{\partial v_{x_{M,M}}} \right]^T \tag{3.17}$$

The second partial derivative of T_2 is computed as

$$\frac{\partial^2 T_2}{\partial v_{x_{p,q}} \partial v_{x_{k,m}}} = \begin{cases} \frac{2}{2|\tilde{v}_{x_{p,q}} - \tilde{v}_{x_{p-1,q}}| + \epsilon} + \frac{2}{2|\tilde{v}_{x_{p+1,q}} - \tilde{v}_{x_{p,q}}| + \epsilon} & \text{if } p = k \text{ and } q = m \\ -\frac{2}{2|\tilde{v}_{x_{p,q}} - \tilde{v}_{x_{p-1,q}}| + \epsilon} & \text{if } p = k+1 \text{ and } q = m \\ -\frac{2}{2|\tilde{v}_{x_{p+1,q}} - \tilde{v}_{x_{p,q}}| + \epsilon} & \text{if } p = k-1 \text{ and } q = m \\ 0 & \text{others} \end{cases} \tag{3.18}$$

The Hessian matrix $(N \times N)$ of T_2 can be formed as

$$T_{2_{Hess}} = \begin{bmatrix} \frac{\partial^2 T_2}{\partial v_{x_{1,1}}^2} & \frac{\partial^2 T_2}{\partial v_{x_{1,1}} \partial v_{x_{1,2}}} & 0 & \cdots & 0 & 0 \\ \frac{\partial^2 T_2}{\partial v_{x_{1,2}} \partial v_{x_{1,1}}} & \frac{\partial^2 T_2}{\partial v_{x_{1,2}}^2} & \frac{\partial^2 T_2}{\partial v_{x_{1,2}} \partial v_{x_{1,3}}} & \cdots & 0 & 0 \\ 0 & \frac{\partial^2 T_2}{\partial v_{x_{1,3}} \partial v_{x_{1,2}}} & \frac{\partial^2 T_2}{\partial v_{x_{1,3}}^2} & \cdots & 0 & 0 \\ 0 & 0 & \frac{\partial^2 T_2}{\partial v_{x_{1,4}} \partial v_{x_{1,3}}} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & \frac{\partial^2 T_2}{\partial v_{x_{M,M-1}}^2} & \frac{\partial^2 T_2}{\partial v_{x_{M,M-1}} \partial v_{x_{M,M}}} \\ 0 & 0 & 0 & \cdots & \frac{\partial^2 T_2}{\partial v_{x_{M,M}} \partial v_{x_{M,M-1}}} & \frac{\partial^2 T_2}{\partial v_{x_{M,M}}^2} \end{bmatrix} \tag{3.19}$$

It is easily observed that $T_{2_{Hess}}$ is a symmetric, positive definite matrix.

3.3.3 Iterative Algorithm

Given the Jacobian vectors and Hessian matrixes of the two terms, respectively, the Jacobian and Hessian of objective T_v is the summation of their Jacobian and Hessian of terms T_1 and T_2, i.e.

$$T_{Jac} = T_{1_{Jac}} + \lambda T_{2_{Jac}} \tag{3.20}$$

$$T_{Hess} = T_{1_{Hess}} + \lambda T_{2_{Hess}} \tag{3.21}$$

From Reference [2] and Equation 2.10, we have

$$T_{Jac} = -T_{Hess}\delta v_x \tag{3.22}$$

Obviously, if T_{Hess} is non-singular, δv_x can be obtained by solving the above linear system.

The iterative algorithm can be summarized as below:

1. Set number of iterations and initialization of velocity matrix by $M \times M$.
2. Calculate $T_{1_{Jac}}$, $T_{1_{Hess}}$, $T_{2_{Jac}}$ and $T_{2_{Hess}}$ by using Equations 3.11, 3.13, 3.17 and 3.19.
3. Solve the velocity change by using Equation 3.22.
4. Update the velocity as
$$v_x^{k+1} = v_x^k + \delta v_x^k \tag{3.23}$$
5. Repeat 2-4 until reaching the number of iterations.

3.4 Simulation with MATLAB

We have developed the iterative algorithm for our specific problem–MRI velocity quantification. The algorithm is simulated with MATLAB. The MATLAB code can be found in Appendix B.

To investigate how the noise, λ, and G affect the solution, the experiments for different cases have been carried out. For the sake of comparison, we define a criterion to measure the error between the true velocity v_x^* and the estimated velocity v_x in norm 2.

$$Error = ||v_x - v_x^*||_2 \tag{3.24}$$

Case 1: $G = 0.01$ and real noise for different λ

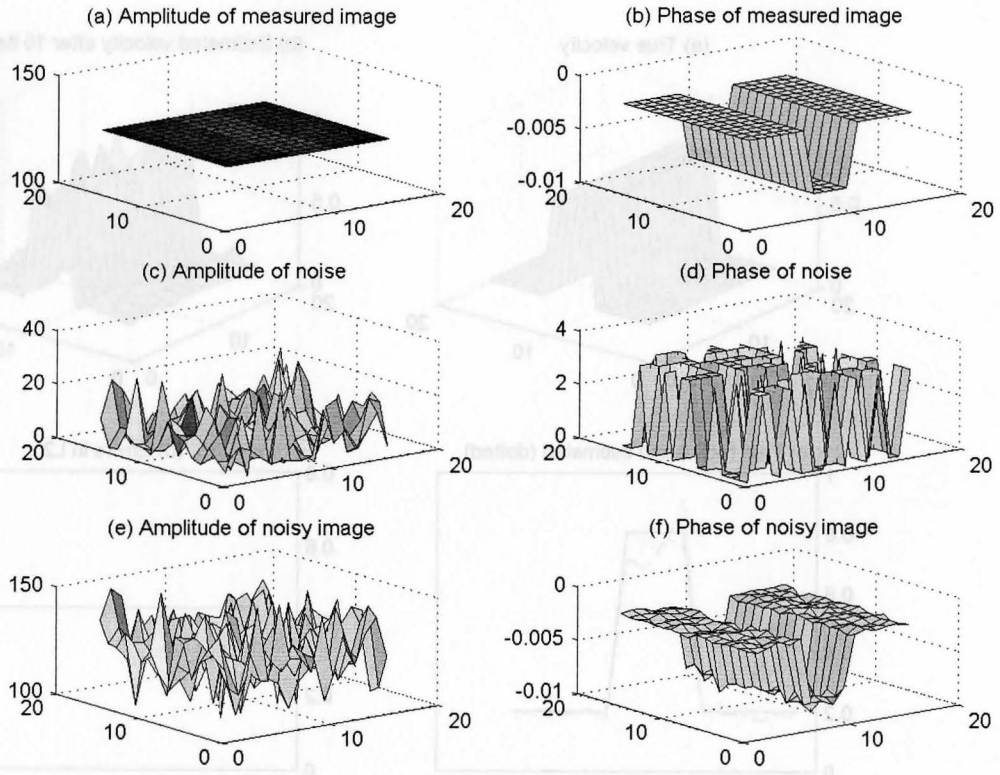

Figure 3.1: Measured images and real noise

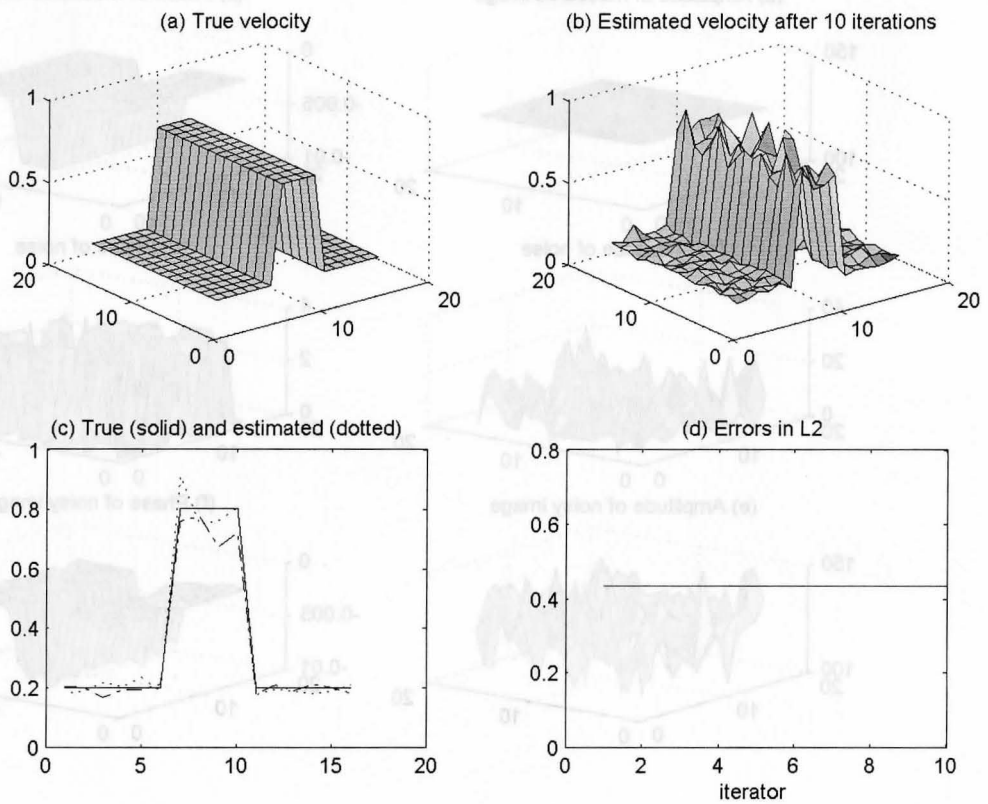

Figure 3.2: Velocity estimation with real noise ($\lambda = 0$)

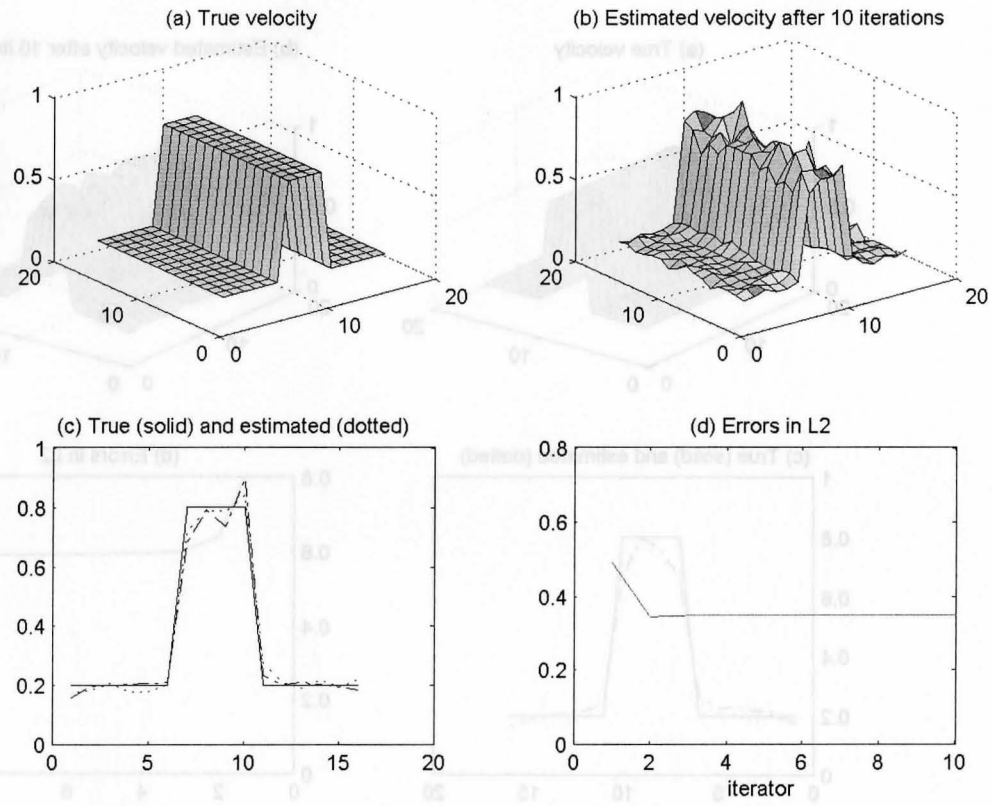

Figure 3.3: Velocity estimation with real noise ($\lambda = 0.2$)

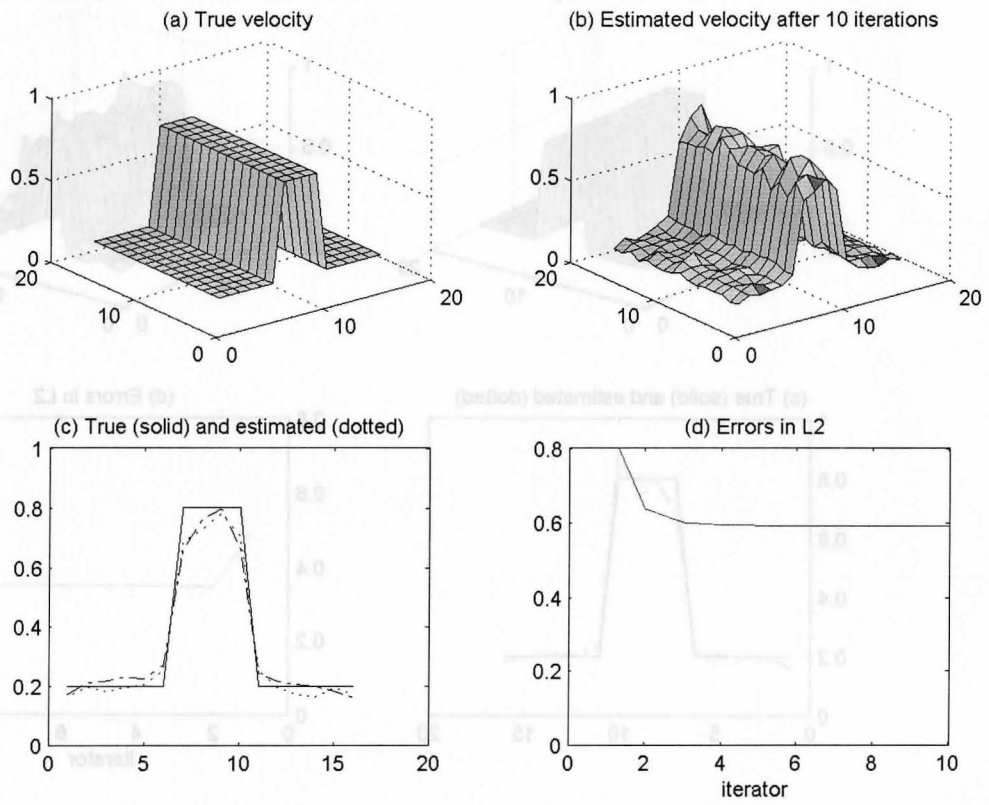

Figure 3.4: Velocity estimation with real noise ($\lambda = 1$)

Fig. 3.1 shows the amplitude and phase of the images and noise, respectively. From 3.1b, the velocity only affects the phase of the measure image. Without TV ($\lambda = 0$), the estimated velocity with the measure image shown in Figs. 3.1e-3.1f is presented in Fig. 3.2. It is seen that there are sharp changes shown in Figs.3.2b and 3.2c. Figs. 3.3 and 3.4 demonstrate the estimated velocity when $\lambda = 0.2$ and $\lambda = 1$, respectively. Comparing Fig. 3.2b with Figs. 3.3b-3.4b and Fig. 3.2d with 3.3d-3.4d, it is observed that the total variation term with a regularization parameter does smooth the solution. It is also found that $\lambda = 0.2$ is the optimal regularization parameter with the minimum stable error and smooth solution, compared to the cases of $\lambda = 0$ and $\lambda = 1$. When $\lambda = 1$, the solution is too smooth and causes a large stable error.

The examples show that the algorithm works well for the real noise. However, when a complex noise is added, the phase of the noise dominates the phase of the noisy image if we use the small G. The algorithm does not work for such a small G. We have to choose a large G.

Case 2: $G = 1$ and complex noise for different λ

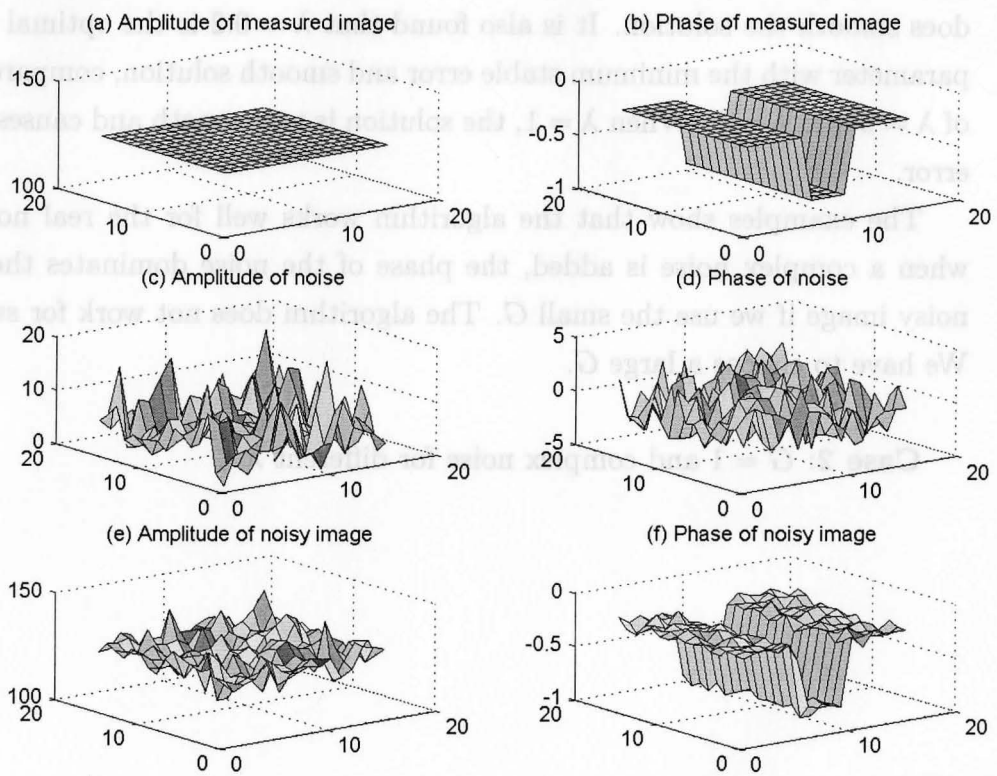

Figure 3.5: Measured images and complex noise

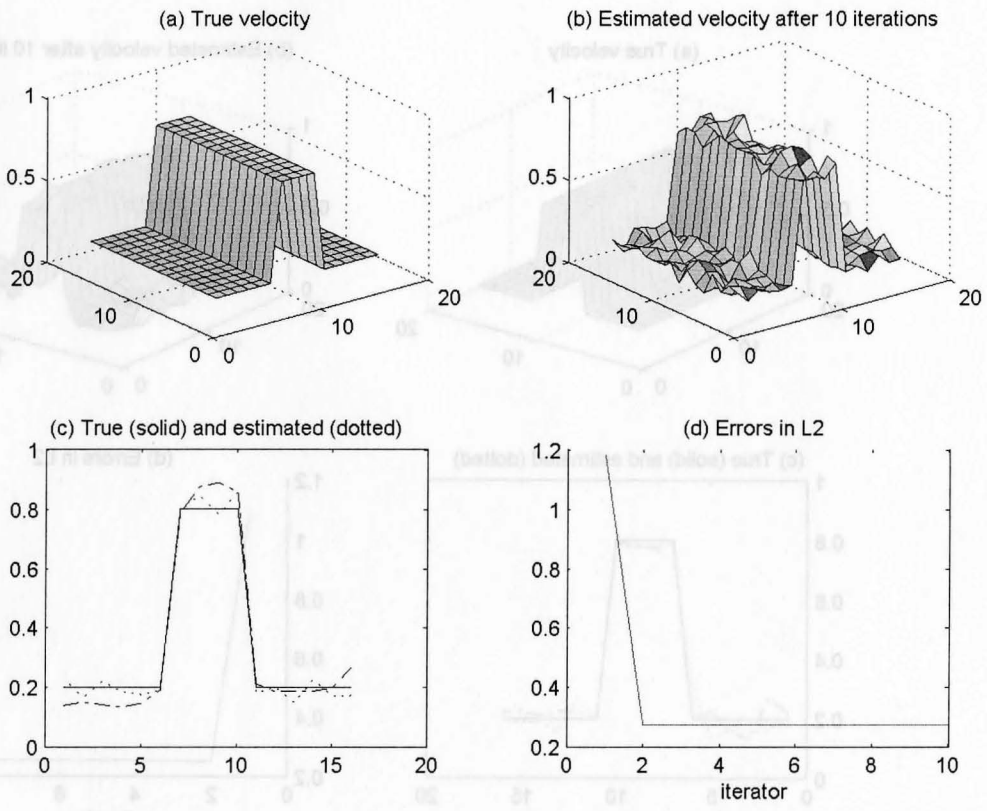

Figure 3.6: Velocity estimation with complex noise ($\lambda = 0$)

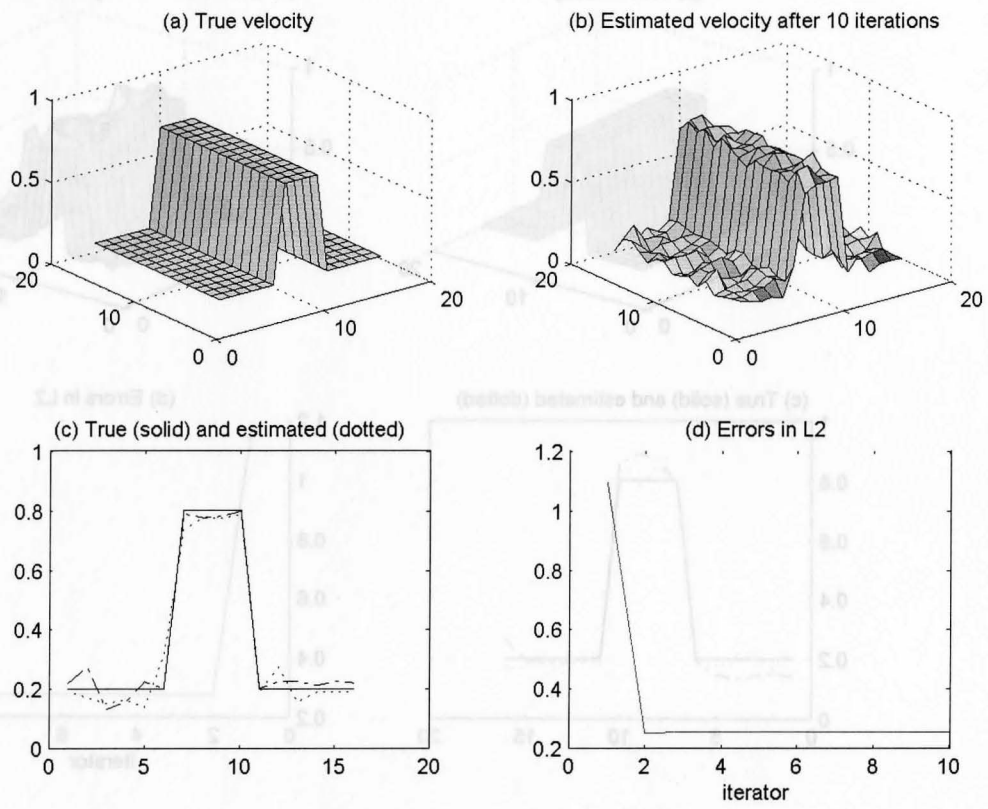

Figure 3.7: Velocity estimation with complex noise ($\lambda = 1000$)

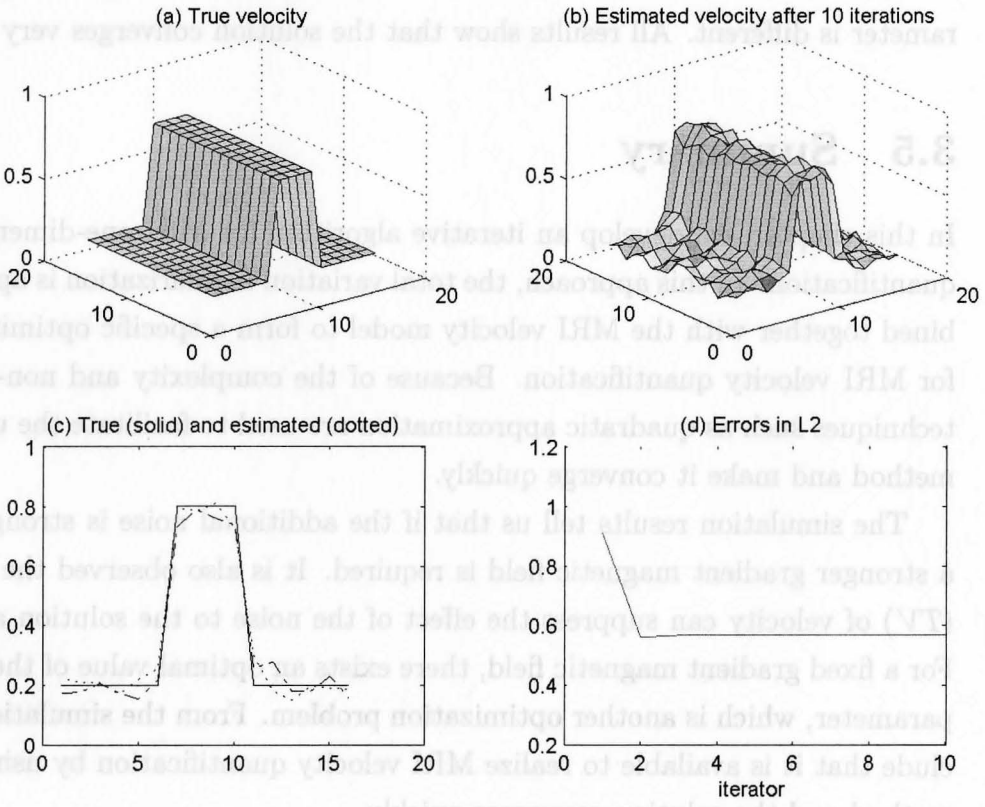

Figure 3.8: Velocity estimation with complex noise ($\lambda = 5000$)

From Fig. 3.5, it is shown that a strong complex noise is applied to the system. According to the results shown in Figs. 3.6-3.8, the TV of velocity does smooth the solution of velocity by taking a big regularization parameter when the complex noise is present. Similar to the Case 1, the solution with $\lambda = 1000$ is superior to the ones with $\lambda = 0$ and $\lambda = 5000$.

From the two cases of noises, it is easily found that the good regularization parameter is different. All results show that the solution converges very quickly.

3.5 Summary

In this chapter, we develop an iterative algorithm for MRI one-dimensional velocity quantification. In this approach, the total variation regularization is applied and combined together with the MRI velocity model to form a specific optimization problem for MRI velocity quantification. Because of the complexity and non-linearity, some techniques such as quadratic approximation are used to facilitate the use of Newton's method and make it converge quickly.

The simulation results tell us that if the additional noise is strong and complex, a stronger gradient magnetic field is required. It is also observed the total variation (TV) of velocity can suppress the effect of the noise to the solution at some extent. For a fixed gradient magnetic field, there exists an optimal value of the regularization parameter, which is another optimization problem. From the simulation, we can conclude that it is available to realize MRI velocity quantification by using the iterative method and the solution converges quickly.

Chapter 4

Algorithm Implementation with CBE

4.1 Features of DMA Transfer with CBE

As we described Chapter 1, the Cell Broadband Engine is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. One is the PPE, is a 64-bit PowerPC Architecture core. It is fully compliant with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The other processor with the eight SPEs is optimized for running compute-intensive applications, and it is not optimized for running an operating system. The SPEs are independent processors, each running its own individual application programs. Each SPE has full access to coherent shared memory, including the memory-mapped I/O space. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level control thread of an application. The PPE depends on the SPEs to provide the bulk of the application performance.

A significant difference between the SPEs and the PPE is how they access memory. The PPE accesses main storage (the effective-address space that includes main memory) with load and store instructions that go between a private register file and main storage (which may be cached). However, the SPEs access main storage with

direct memory access (DMA) commands that go between main storage and a private local memory used to store both instructions and data. SPE instruction-fetches and load and store instructions access this private local store, rather than shared main storage.

One type of programming model may rely on the PPE to perform the task of application management by assigning and distributing work to the SPEs. A significant part of this task may be loading main storage with programs and data and then notifying an SPE of available work by either writing to its mailbox or one of its signal-notification registers. After getting the message or signal, the SPE performs a DMA operation to transfer the data and code to its LS. In a variation on this programming model, the PPE might perform the DMA operation and then send a message or signal to the SPE when the DMA operation completes.

After processing the data, an SPE can use another DMA operation to deliver the results to main storage. When the DMA transfer of the SPE results from LS to main storage finishes, the SPE can write a completion message to one of its two outgoing mailboxes that informs the PPE that processing and delivery is complete. If the completion message requires more than 32 bits of information, the SPE can write multiple mailbox messages or use a DMA operation to transfer the long message to main storage where the PPE can read it. Even when a long completion message is transferred with a DMA operation, an outgoing mailbox message can be used to inform the PPE that the message is available.

The communication between PPE with SPEs and between SPEs is controlled by the SPE's Memory Flow Controller (MFC). The majority of MFC commands initiate DMA transfers. These are called DMA commands, i.e., the basic get and put DMA commands.

When programming with PPE and SPE, the following factors should be considered:

Size of DMA Transfers DMA transfers move up to 16 KB of data between the LS of an SPE and main storage. An MFC supports naturally aligned DMA transfer sizes of 1, 2, 4, 8, and 16 bytes and multiples of 16 bytes.

Limitation of Local Store Each LS is limited to 256 KB for program, stack, local data structures, and DMA buffers. Many optimization techniques put pressure on this limited resource. As such, all optimizations may not be possible for a given application. Often it is possible to reduce LS pressure by dynamically managing the program store using code overlays.

4.2 Algorithm Development for Solving Linear Equations

For our application, MRI velocity quantification, the image size varies from small, e.g. 16x16, to large, e.g., 256x256. The proposed algorithm in Chapter 3 requires computation of Hessian, which is a quite big matrix. For instance, with an image size of 16x16, the Hessian matrix will be 256×256, 256 times larger than the image size. In general, if the image size is $M \times M$, the Hessian is $(M \times M) \times (M \times M)$ with M^2 times larger than the image size.

4.2.1 Direct Implementation of the Algorithm

At the first place, the algorithm is directly implemented with PPU and SPU without any modifications. The flowcharts of the implementation for PPU and SPU are shown in Figures 4.1 and 4.2, respectively. Note that in the PPU flowchart, the flag in the SPU status function call will tell if the SPU finished with 1 returned if SPU finished.

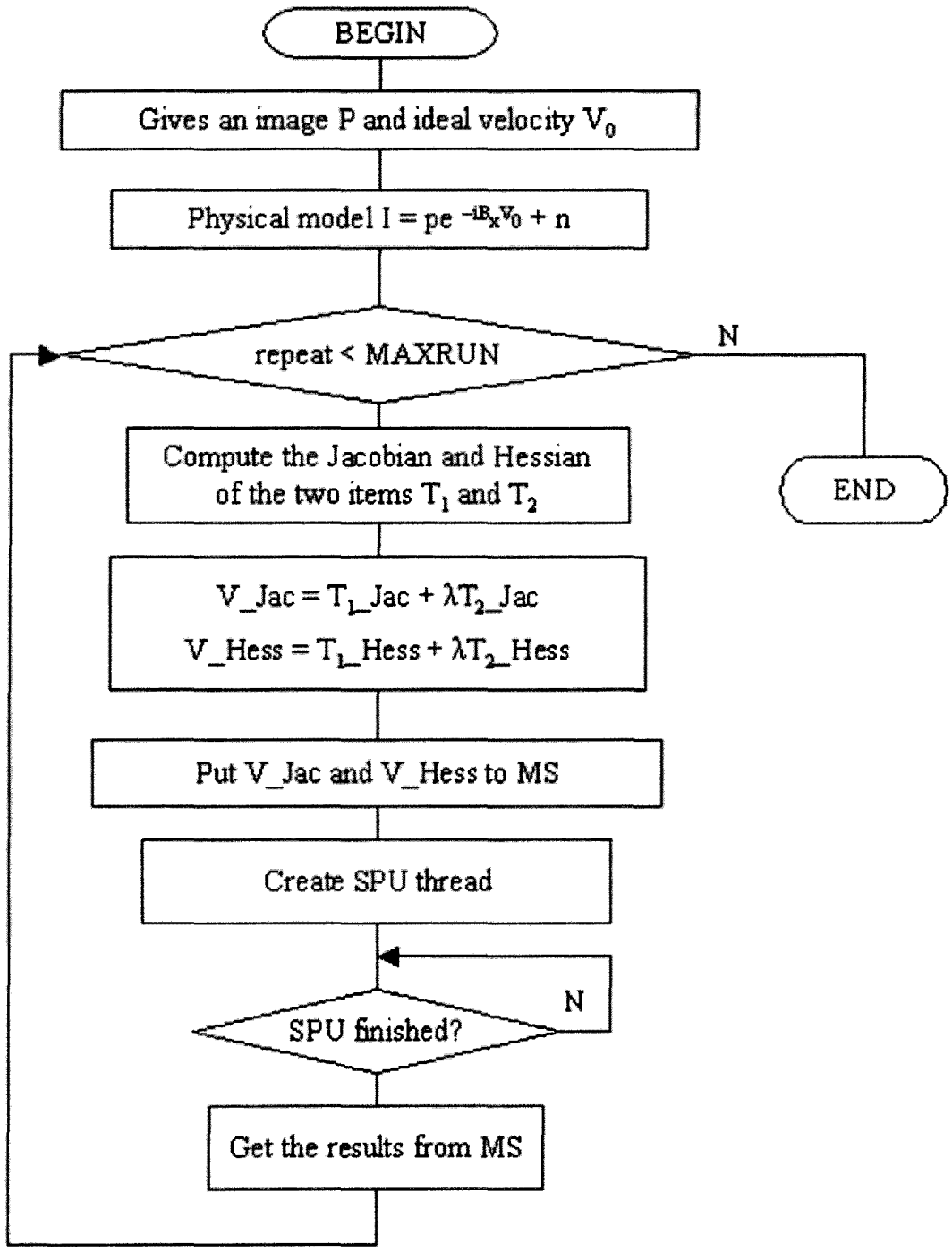

Figure 4.1: PPU Flowchart of Velocity Quantification Implementation

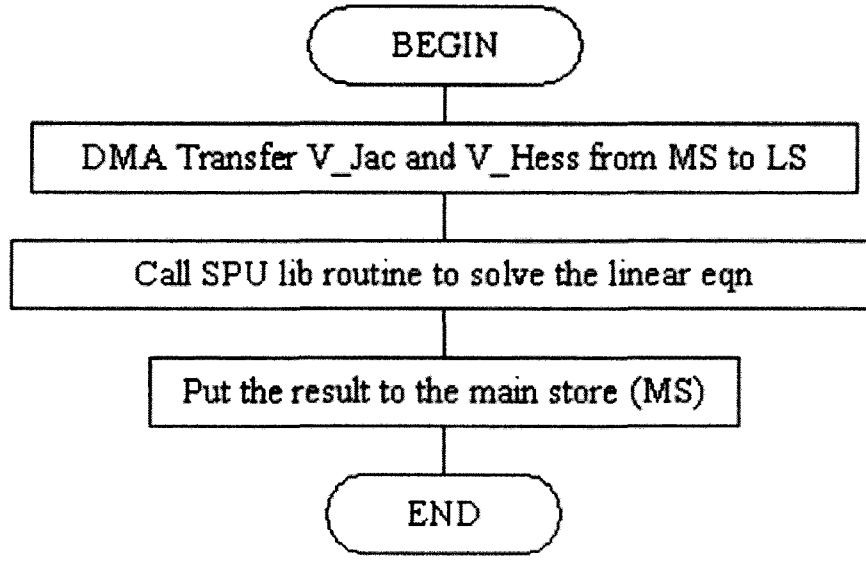

Figure 4.2: SPU Flowchart of Velocity Quantification Implementation

We found out that the maximum size of image allowed is 8x8 in the algorithm. Otherwise, a bus error will be reported. In most applications, this size seems too small, resulting the algorithm is very limited in practice.

Why does it happen? By calculating the size of data and the code for an image 12x12, it does not exceed the limit of the LS. However, for a Hessian of 144x144 with floating point numbers, the size of DMA transfers of the Hessian only in the implementation is 81KB (>>16KB). This is the reason why the algorithm failed even for a small size image, 12x12. Of course, the limitation of LS is another issue to be considered for a larger image.

To make this algorithm more effective in practice and to accommodate the algorithm to the CBE architecture, we have to modify the algorithm.

4.2.2 Algorithm Development for Solving Linear Equations

Let us first rewrite the linear equation.

$$T_{Jac} = -T_{Hess}\delta v_x$$

In particular, the structure of T_{Jac}, T_{Hess} and δv_x are given as below:

$$T_{Hess} = \begin{bmatrix} a_0 & c_0 & 0 & 0 & \cdots & 0 & 0 \\ c_0 & a_1 & c_1 & 0 & \cdots & 0 & 0 \\ 0 & c_1 & a_2 & c_2 & \cdots & 0 & 0 \\ 0 & 0 & c_2 & a_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & \cdots & c_{n-2} & a_{n-1} \end{bmatrix},$$

$$T_{Jac} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{bmatrix}, \delta v_x = \begin{bmatrix} \delta v_0 \\ \delta v_1 \\ \delta v_2 \\ \delta v_3 \\ \vdots \\ \delta v_{n-2} \\ \delta v_{n-1} \end{bmatrix}$$

where $n = (M \times M)$ and $M \times M$ is the image size.

By moving the sign symbol into the vector δv_x, we have

$$\begin{bmatrix} a_0 & c_0 & 0 & 0 & \cdots & 0 & 0 \\ c_0 & a_1 & c_1 & 0 & \cdots & 0 & 0 \\ 0 & c_1 & a_2 & c_2 & \cdots & 0 & 0 \\ 0 & 0 & c_2 & a_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & \cdots & c_{n-2} & a_{n-1} \end{bmatrix} \begin{bmatrix} \delta v_0 \\ \delta v_1 \\ \delta v_2 \\ \delta v_3 \\ \vdots \\ \delta v_{n-2} \\ \delta v_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-2} \\ b_{n-1} \end{bmatrix} \tag{4.1}$$

By observing the structure of T_{Hess}, it is easily to find that T_{Hess} is a sparse matrix with a certain pattern. With this structure, the Cholesky decomposition can

be adopted. Hereafter, we attempt to propose a simple method to solve the linear equations with the same matrix pattern as T_{Hess}.

By making use of linear transform, i.e., eliminating one variable by mathematical manipulation from every two adjacent equations, we obtain

$$\begin{bmatrix} a_0 & c_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & a'_1 & c_1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & a'_2 & c_2 & \cdots & 0 & 0 \\ 0 & 0 & 0 & a'_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a'_{n-2} & c_{n-2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & a'_{n-1} \end{bmatrix} \begin{bmatrix} \delta v_0 \\ \delta v_1 \\ \delta v_2 \\ \delta v_3 \\ \vdots \\ \delta v_{n-2} \\ \delta v_{n-1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_{n-2} \\ b'_{n-1} \end{bmatrix} \tag{4.2}$$

Comparing the new matrix with T_{Hess}, we observe that the elements in main diagonal are changed except for the first element and all elements under the main diagonal are zeros. There is no changes for the elements in upper triangle. The elements of T_{Jac} are also modified except for the first element. The updated elements are given as below:

$$\begin{cases} a'_1 = a_1 - \frac{c_0^2}{a_0} & b'_1 = b_1 - \frac{c_0 b_0}{a_0} \\ a'_2 = a_2 - \frac{c_1^2}{a'_1} & b'_2 = b_2 - \frac{c_1 b'_1}{a'_1} \\ \vdots & \vdots \\ a'_{n-1} = a_{n-1} - \frac{c_{n-2}^2}{a'_{n-2}} & b'_{n-1} = b_{n-1} - \frac{c_{n-2} b'_{n-2}}{a'_{n-2}} \end{cases} \tag{4.3}$$

From Equation 4.2, the elements of δv_x can be easily calculated back-substitutive.

$$\begin{cases} \delta v_{n-1} = \frac{b'_{n-1}}{a'_{n-1}} \\ \\ \delta v_{n-2} = (b'_{n-2} - c_{n-2}\delta v_{n-1})\frac{1}{a'_{n-2}} \\ \\ \delta v_{n-3} = (b'_{n-3} - c_{n-3}\delta v_{n-2})\frac{1}{a'_{n-3}} \\ \vdots \end{cases} \tag{4.4}$$

Therefore, the implementation of the part of solving the linear equation with CBE SPU can be simplified in the following:

1. DMA transfer three sets, $\{a_0, a_1, \cdots, a_{n-1}\}$, $\{b_0, b_1, \cdots, b_{n-1}\}$, $\{c_0, c_1, \cdots, c_{n-2}\}$
2. Updating the elements in formula 4.3
3. Calculating the elements of δv_x one by one in formula 4.4

The flowchart of the PPU implementation is similar to the one presented in Figure 4.1. The only difference is that the only non-zeros elements (diagonal and subdiagonal) of the Hessian are transferred to the LS. The flowchart of the SPU implementation is shown in Figure 4.3.

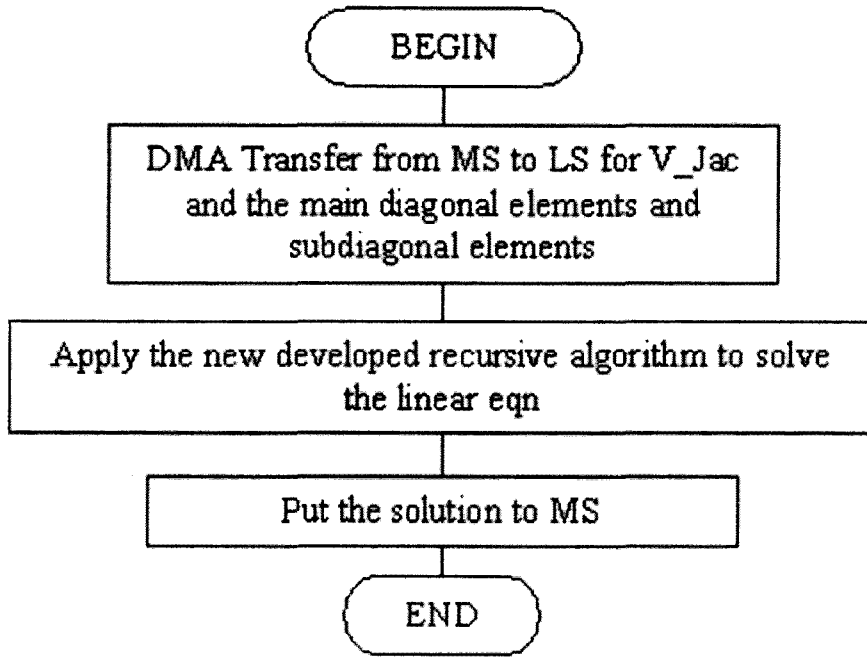

Figure 4.3: SPU Flowchart of New Algorithm for Velocity Quantification Implementation

4.2.3 Image Size

Using the new developed algorithm, the size of the DMA transfers can be largely reduced from $(M \times M) \times (M \times M) + M \times M$ to $3(M \times M) - 1$. For an image with size of $M \times M$ with floating point format, we can derive the upper bound of image size by considering the limitation of DMA transfers.

The original algorithm

$$4(M^4 + M^2) \leq 16 \times 254 \times 4 \Rightarrow M \leq 8 \tag{4.5}$$

The new algorithm

$$12M^2 \leq 16 \times 254 \times 4 \Rightarrow M \leq 36 \tag{4.6}$$

With the new algorithm, the maximum image size is increased from 8x8 to 36x36 for the algorithm in CBE SPU.

4.3 Experiments with CBE

The objective of the experiments is, on one hand, to verify the estimated velocity being consistent with the ones from the results from MATLAB and, on the other hand, to quantify performance and identify issues on the PPU and SPU.

First, let us describe the experiment environment.

- Machine for compiling: Machine 1 with SDK1.1 installed
- Machine for execution: Machine 2 with CBE installed
- Compilers: IBM latest XLC compilers, spuxlc and ppuxlc, for SPU and PPU respectively
- Implementation language: C and SPU intrinsics

All experiments with CBE used the same input data as used with MATLAB experiment shown in last chapter.

Verification of the New Algorithm Two versions of the new algorithm are implemented. One is the PPU version, which has no SPU code. Another one includes PPU code and SPU code, which is a normal way to program with CBE.

The PPU version run on main memory and does not need DMA transfer. The Image size is not limited to 36x36. The experiments show that the algorithm still works for the image size of 256x256. It is easy to understand that the image size is limited only by the available memory for the PPU version.

The combined PPU & SPU version means that the PPU reads image as input, and carries out simple tasks and creates threads for SPU. The SPU gets data through DMA transfer and performs the required computation. This implementation is limited by size of Local Store (LS) and size of DMA transfer. As explained in the last section, the maximum image size allowed for the new algorithm is 36x36.

From the experiments, the estimated velocity from both versions consistent with the results from the ones from MATLAB simulation. Here we do not repeat the figures again. See the figures in Simulation Section in Chapter 3.

Performance Comparison The two versions were run on Machine 2 10 times and the time elapsed was recored in Table 4.1. From the table, it is found that the ticks vary with tests. This is due to the fact that more than one user was using the machine. It is also observed that the combined PPU & SPU code performs 10 times faster than the PPU code though only scalar form is used. The experiments confirm that the SPU is optimized for compute-intensive applications of this type.

Table 4.1: PPU and SPU Performance Comparison

Test Number	SPU & PPU (ticks)	PPU (ticks)	PPU/SPU ratio
1	436	4450	10.2
2	440	4528	10.3
3	428	4339	10.1
4	453	4724	10.4
5	438	4486	10.2
6	450	4648	10.3
7	446	4562	10.2
8	459	4736	10.3
9	432	4420	10.2
10	448	4625	10.3
Avg.	443	4551.8	10.3

4.4 Summary

In this chapter, we focus on the algorithm implementation with CBE. Based on the algorithm proposed in Chapter 3, the direct implementation was first performed by calling the subroutine in SDK1.1 for solving linear equations. The image size allowed, however, is very small because of the limitation of LS and DMA transfers. To solve this problem, a new algorithm is proposed, which allows maximum image size up to 36x36.

Two versions (PPU and combined PPU & SPU) of implementation were produced. The experiments showed that both versions have the same results as the ones from MATLAB. Meanwhile, the combined PPU & SPU method run 10 times faster than the PPU version though the PPU has more flexibility on image size.

Part II

Floating Point Math Functions Test with CBE

Chapter 5

Single-Precision Floating Point Numbers

The floating-point representation is widely used for representing real numbers. It represents reals in scientific notation, which has a base number and an exponent. IEEE Standard 754 floating point is the most common used representation for real numbers on computers, including Intel-based PC's, Macintoshes, and most Unix platforms. This chapter will give a brief overview of IEEE Standard 754 floating point representation with single-precision. The concepts to be introduced will help understand the test on elementary math functions with single-precision for CBE SPU in the next chapter.

5.1 Storage Formats

A floating-point storage format specifies how a floating-point format is stored in memory. The IEEE standard defines the formats, but it leaves to implementors the choice of storage formats [32]. IEEE 754 is a binary standard and the format consists of three components: the sign, the exponent, and the mantissa. The mantissa is composed of the fraction and an implicit leading digit. The exponent base (2) is implicit and does not need to be stored [33]

Table 5.1 shows the storage layout for single-precision (32-bit) floating-point num-

ber and the number of bits for each field. The bit ranges are in square brackets.

Table 5.1: IEEE Single-Precision Format

Sign	Exponent	Fraction	Bias
1[31]	8 [30-23]	23 [22-00]	127

- **The Sign Bit**: 0 refers to a positive number; 1 denotes a negative number.
- **The Exponent**: The exponent field needs to represent both positive and negative exponents. To do this, a bias is added to the actual exponent in order to get the stored exponent. For IEEE single-precision floats, this value is 127. Thus, an exponent of zero means that 127 is stored in the exponent field. A stored value of 157 indicates an exponent of (157-127), or 30 [33].
- **The Mantissa**: The mantissa, also known as the significand, represents the precision bits of the number. It is composed of an implicit leading bit and the fraction bits. In order to maximize the quantity of representable numbers, floating-point numbers are typically stored in normalized form. This basically puts the radix point after the first non-zero digit. We can just assume a leading digit of 1, and do not need to represent it explicitly. As a result, the mantissa has effectively 24 bits of resolution, by means of 23 fraction bits. The mantissa can be represented as $1.f$, where f is the fraction bits [33].

5.2 Ranges of Single-Precision Floating-Point Numbers

The range of single-precision floating point numbers can be split into normalized numbers with the full precision of the mantissa, and denormalized numbers with a portion of the fractions's precision, shown in Table 5.2.

From this table, it is found that the single-precision representation cannot represent five numerical ranges.

Table 5.2: IEEE Single-Precision Range

Denormalized	Normalized	Approximate Decimal
$\pm 2^{-149}$ to $(1 - 2^{-23}) \times 2^{-126}$	$\pm 2^{-126}$ to $(2 - 2^{-23}) \times 2^{127}$	$\pm\ 10^{-44.85}$ to $\pm\ 10^{38.53}$

1. Negative numbers less than $-(2 - 2^{-23}) \times 2^{127}$ (negative overflow)
2. Negative numbers greater than -2^{-149} (negative underflow)
3. Zero
4. Positive numbers less than 2^{-149} (positive underflow)
5. Positive numbers greater than $(2 - 2^{-23}) \times 2^{127}$ (positive overflow)

Note that overflow means that values are too large for the representation and underflow means that values are too small to represent. Underflow is a less serious problem because it just denotes a loss of precision, which is guaranteed to be closely approximated by zero [33]. The maximum positive normal number is the largest finite number representable in IEEE single format. The minimum positive denormalized number is the smallest positive number representable in IEEE single format.

5.3 Special Quantities and Operations

IEEE reserves exponent field values of all 0s and all 1s to denote special values in the floating-point scheme.

Zero As mentioned above, zero is not directly representable in the straight format, due to the assumption of a leading 1 (we'd need to specify a true zero mantissa to yield a value of zero). Zero is a special value denoted with an exponent field of zero and a fraction field of zero. Note that -0 and +0 are distinct values, though they both compare as equal.

Denormalized If the exponent is all 0s, but the fraction is non-zero (else it would be interpreted as zero), then the value is a denormalized number, which does not have an assumed leading 1 before the binary point. Thus, this represents a number $(-1)^s \times 0.f \times 2^{-126}$, where s is the sign bit and f is the fraction. Zero can be interpreted as a special type of denormalized number.

Infinity The values $+infinity$ and $-infinity$ are denoted with an exponent of all 1s and a fraction of all 0s. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations. Operations with infinite values are well defined in IEEE floating point.

Not A Number The value NaN (Not a Number) is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.
Table 5.3 shows the bit patterns for the special quantities and a normalized real. *Note*: b means the IEEE bias and X means any value of 0 or 1.

Table 5.3: Bit Patterns and the IEEE Values in Single-Precision Format

Sign (s)	**Exponent (e)**	**Fraction (f)**	**Value**
0 or 1	$0 \cdots 0$	$0 \cdots 0$	$+0$ or -0
0 or 1	$0 \cdots 0$	nonzero	Denormalized Real $(-1)^s \times 0.f \times 2^{-b+1}$
0 or 1	nonzero	X $\cdots$ X	Normalized Real $(-1)^s \times 1.f \times 2^{e-b}$
0 or 1	$1 \cdots 1$	$0 \cdots 0$	$\pm\infty$
0 or 1	$1 \cdots 1$	nonzero	NaN

Special Operations IEEE standard also defines special operations on the special numbers. See Table 5.4.

Table 5.4: Special Operations

Operation	Result
$n \div \pm\infty$	0
$\pm\infty \times \pm\infty$	$\pm\infty$
$\pm nonzero \div 0$	$\pm\infty$
$\infty + \infty$	∞
$\pm 0 \div \pm 0$	NaN
$\infty - \infty$	NaN
$\pm\infty \div \pm\infty$	NaN
$\pm\infty \times 0$	NaN

Note: n refers to a real number.

5.4 Rounding Error and Ulps

Squeezing infinitely many real numbers into a finite number of bits requires an approximate representation. Although there are infinitely many integers, in most programs the result of integer computations can be stored in 32 bits. In contrast, given any fixed number of bits, most calculations with real numbers will produce quantities that cannot be exactly represented using that many bits. Therefore the result of a floating-point calculation must often be rounded in order to fit back into its finite representation. This rounding error is the characteristic feature of floating-point computation. See reference [31].

Since rounding error is inherent in floating-point computation, it is important to have a way to measure this error. There are two ways to measure the error. One is in ulps (shorthand for "units in the last place"). If the result of a calculation is the floating-point number nearest to the correct result, it still might be in error by as much as 0.5 ulp. For example, if the real number .0314159 is represented as

3.14×10^{-2}, then it is in error by .159 ulps. Another way to measure the difference between a floating-point number and the real number it is approximating is relative error, which is simply the difference between the two numbers divided by the real number.

The representations with the same relative error may have different ulps. The most natural way to measure rounding error is in ulps. For example rounding to the nearest floating-point number corresponds to an error of less than or equal to .5 ulp. However, when analyzing the rounding error caused by various formulas, relative error is a better measure [31].

5.5 CBE SPU Floating-Point Support

The CBE SPU supports both single- and double-precision floating-point operations. For single-precision operations, the range of normalized numbers is extended beyond the IEEE standard. The representable, positive, non-zero numbers range from $e_{min} = 2^{-126}$ to $e_{max} = (2 - 2^{-23}) \times 2^{128}$. If the exact result overflows (that is, if it is larger in magnitude than e_{max}), the rounded result is set to e_{max} with the appropriate sign. If the exact result underflows (that is, if it is smaller in magnitude than e_{min}), the rounded result is forced to zero. A zero result is always a positive zero. See reference [5]

Table 5.5 summarizes the SPE floating-point support for single-precision and compliance with the IEEE Standard 754.

From Table 5.5, we see that single-precision floating-point operations implement IEEE 754 arithmetic with the following extensions and differences:

- Only one rounding mode is supported: round towards zero, also known as truncation.
- Denormal operands are treated as zero, and denormal results are forced to zero.
- Numbers with an exponent of all ones are interpreted as normalized numbers and not as infinity or not-a-number (NaN).

Table 5.5: SPE Single-Precision Floating-Point Support

Features	Floating-Point Bit Pattern	Decimal Value or Remarks
e_{min}	(001)([1.]000...000) (Extended-Range Mode)	1×2^{-126}, 1.2×10^{-38} (Extended-Range Mode)
e_{max}	(255)([1.]111...000) (Extended-Range Mode)	$(2 - 2^{-23}) \times 2^{128}$, 6.8×10^{38} (Extended-Range Mode)
Zero	sign bit: 0 or 1 all bits 0s	only output +0
NaN		treated as normalized numbers
Infinity(Inf)		treated as normalized numbers
Denormal		Treated as 0
Rounding Modes		Round to nearest

5.6 Summary

In this chapter, we introduced the floating-point single-precision format (IEEE 754) and representations of special values. Furthermore, by comparing CBE SPU support for single-precision and IEEE standard 754, the compliant and difference were presented. In one word, CBE SPU has a compliant data format with IEEE 754 but does not support denormals.

Chapter 6

Elementary Math Functions Test With CBE

6.1 Overview

The elementary math functions are widely used in application software development. They usually come together with compilers or operating systems to implement built-in functions of computer languages. They are also provided as stand-alone libraries such as IBM's MASS (Mathematical Acceleration SubSystem) or ESSL (Engineering and Scientific Subroutine Library) [34]. Dr. Anand's research group in McMaster was developing the single precision math functions for CBE SPU. In this chapter, we will describe how to develop the test tools and show some test results for the new developed functions.

Test is an integral component of the software life-cycle model. It usually happens after development and before delivery. There are essentially two types of testing, namely, execution-based testing and nonexecution-based testing. As an example of the nonexcution-based testing, a written specification document can be carefully reviewed and analyzed but cannot be executed [30]. However, if there is an executable code, it is possible to run the code with test cases.

The outputs of the test are usually approximately represented by floating-point with single precision or double precision. The more accurate the outputs, the more

bits are required to represent the results and the more time are needed for executing the math functions. A trade-off between accuracy and performance is often considered by the developer of the math functions. When the math library is released for use, information like accuracy and performance is better provided for the user's convenience, see [34, 35, 36]. It will help the user to make a decision on calling the appropriate math functions. However, testing mathematical functions is not an easy task. It is time-consuming, especially for functions with multiple arguments. This is one of the reasons why the information for accuracy and performance is not included with mathematical functions sometimes.

The test we carried out are mostly for the unary functions (i.e. single argument) with single precision. It is possible to examine and verify the results of the functions on every exponent. With reference to the existing testing tool from IBM, a modification is made to accommodate our test cases. The test includes accuracy test and performance test. Furthermore a comparison between our functions and SDK1.1 SPU math functions is made for both accuracy and performance.

Since the functions are tested with CBE, the test environment must be setup first. The environement-based test code for accuracy and performance are developed and will be described in the following sections. Obviously, this belongs to execution-based testing.

6.1.1 Test Environment

Hardware Environment Basically, we used two machines to do the test, machine 1 with the Fedora Core 4 (FC4) and SDK 1.1 installed and machine 2 having the CBE with 2400GHz CPU installed.

Software Environment On machine 1, with the FC4 and SDK1.1 installed, the IBM latest compilers, spuxlc and ppuxlc, are used to compile the test code. After compilation, an executable code is produced. The executable code can be run on either the simulator or CBE. In our case, the executable code is copied to machine CB and is executed.

6.1.2 The Elementary Math Functions

The tested elementary math functions are grouped as below:

- Trig Functions: sin, cos, tan, sincos
- Inverse Trig Functions: asin, acos, atan, atan2
- Exponents and Logarithms: exp, $\log_2$, log, $\log_{10}$, pow, sqrt, cbrt, qdrt
- Hyperbolic Functions: sinh, cosh, tanh
- Inverse Hyperbolic Functions: asinh, acosh, atanh
- Other Function: div, recip, rqdrt, rsqrt, rcbrt

These math functions are tested in three forms— library, inline and long vector. With the above test environment and the developed 27 math functions, the test strategies and results for accuracy and performance will be described in the following sections.

6.2 Accuracy Test

As its name, accuracy test is to test if the math function correct and how accurate if correct. The general idea for testing accuracy is to apply the inputs covered with the entire floating-point range and distributed evenly over the test range. For unary functions with single precision, it is possible to exhaustive test the whole exponent range from -149 to 128. The accuracy of the new functions is measured in ulps of difference between our test results and reference results in C standard library.

6.2.1 Strategies for Accuracy Testing

The implementation of the accuracy test is divided into two parts because of the architecture of CBE. One is the PPU part and another one is the SPU part.
The program in PPU mainly deals with

- Setting up testing range, preparing the arguments

- Copying the argument into array of structure, creating the SPU thread and getting the results from SPU
- Calling the corresponding to the C standard library routines and returning results
- Simulated the SPU floating-point format such as treating the denormalized data as zeros and treating infinity as maximum number.
- Calculating the errors between the test results and reference results
- Printing the reports for every exponent and statistics

The flowchart of this implementation is shown in Figure 6.1.

In contrast, the SPU code is responsible for

- DMA transfer the data from MS to LS
- Calling the new developed SPU mathematical functions
- DMA transfer the results from LS to MS

The flowchart of this implementation is shown in Figure 6.2.

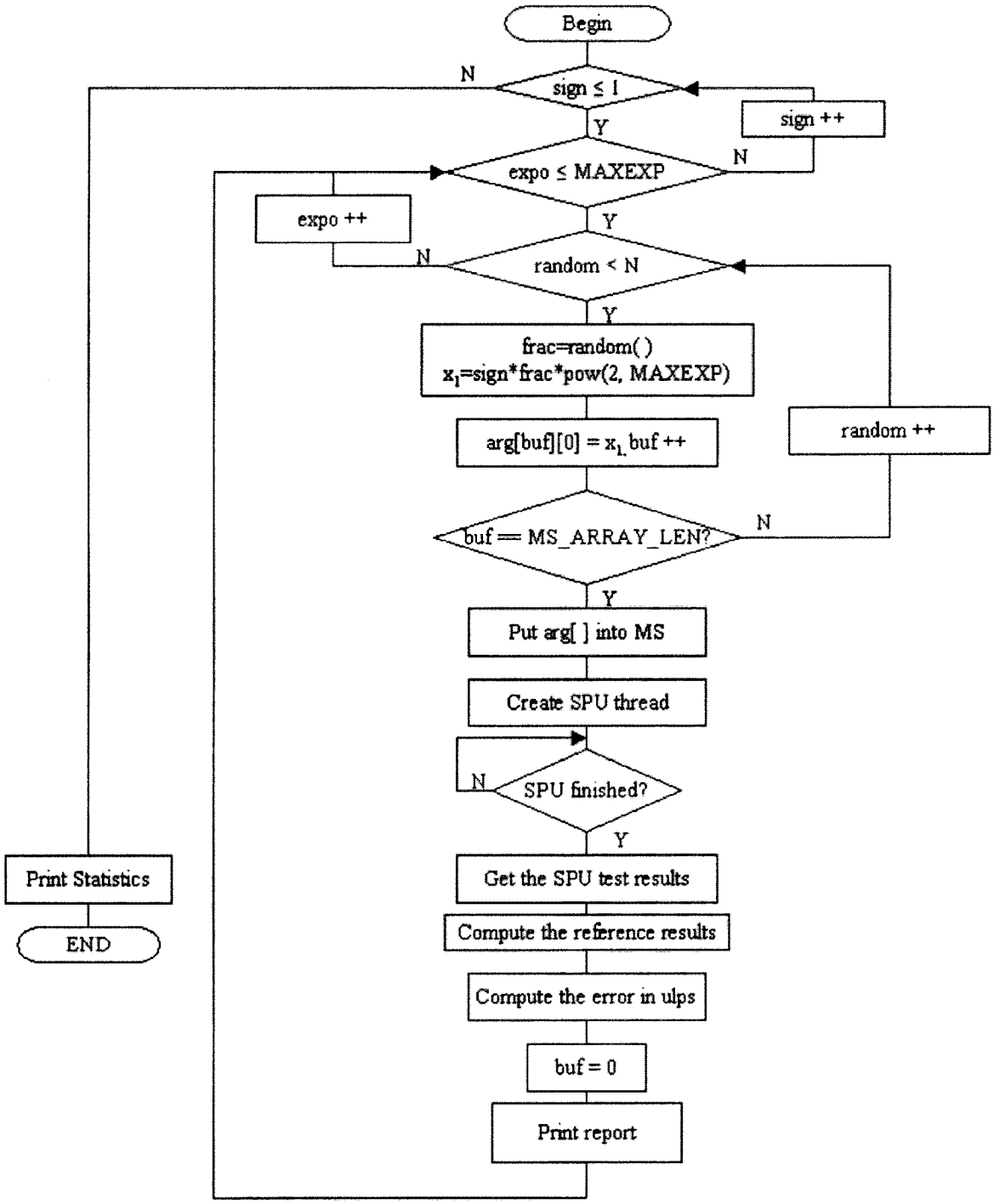

Figure 6.1: PPU Flowchart of Accuracy Test Implementation

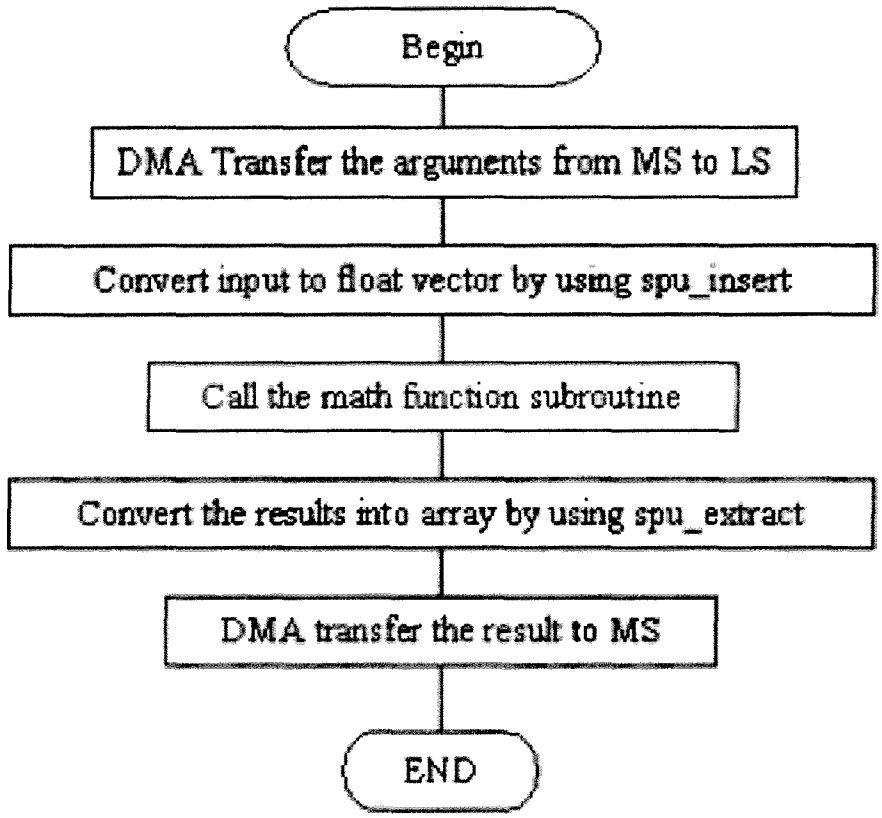

Figure 6.2: SPU Flowchart of Accuracy Test Implementation

6.2.2 Accuracy Test Results

We ran the accuracy test code by calling the math functions with a reasonable arguments range. The output is compared with the corresponding reference value and the difference is represented in ulps. It should be mentioned that different functions have different working range. For example, log function only works for positive arguments.

Part of the accuracy results are shown in Table 6.1 and Table 6.2 in form of histogram. And a comparison between our functions and the existing SDK1.1 functions is displayed in Table 6.3. In these tables, row 2 corresponding to the good range of arguments in decimal and row 3 refers to the maximum errors in ulps and the numbers in rows 4-9 represent the percentage of the error ulps falling into the corre-

sponding range on the left. Obviously, our functions, log and inverse, outperform the log in SDK1.1 from accuracy point of view and the exp function has a little smaller maximum error than the exp in SDK1.1. Because of limited space, we cannot list all the test results. The results will be posted in IBM website.

Note that, as we known, if a software product is executed with test data and the output is wrong, then the product definitely contains a bug. However, if the output is correct, then there still may be a bug in the product. All our test has shown that the product runs correctly on that particular set of test data.

Table 6.1: Accuracy Test Results for the New Functions

	sinh	cosh	tanh	rcbrt
Range	(-32 32)	(-32 32)	full	full> 0
Max ulperr	47.58	32.78	7.88	3.66
[0, 0.5]	93.6	84.9	92.0	49.6
(0.5, 1]	3.1	12.7	6.1	31.8
(1, 2]	1.8	1.0	1.6	17.0
(2, 5]	0.5	0.6	0.3	1.7
(5, 10]	0.4	0.4	0.0	0.0
> 10	0.6	0.3	0.0	0.0

Table 6.2: Accuracy Test Results for the New Functions

	asinh	acosh	atanh	qdrt	rqdrt
Range	$(-2^{127}\ 2^{127})$	full≥ 0	full	full≥ 0	full≥ 0
Max ulperr	3.77	12.36	4.30	4.31	2.20
[0, 0.5]	83.2	67.0	92.3	15.4	74.9
(0.5, 1]	14.7	28.0	6.7	22.3	22.1
(1, 2]	2.1	4.9	1.0	48.7	2.9
(2, 5]	0.1	0.1	0.0	13.6	0.0
(5, 10]	0.0	0.0	0.0	0.0	0.0
> 10	0.0	0.0	0.0	0.0	0.0

Table 6.3: Accuracy Comparison with SDK1.1

	logSDK	log	expSDK	exp	inverseSDK	recip
Range	full> 0	full> 0	(-4 4)	(-4 4)	full	full
Max ulperr	2030.87	3.38	6.63	4.57	1.99	1.44
[0, 0.5]	0.2	72.2	78.9	69.4	31.1	48.1
(0.5, 1]	1.9	25.2	16.6	25.2	39.0	45.3
(1, 2]	53.2	2.5	4.1	5.1	30	6.6
(2, 5]	44.4	0.1	0.4	0.2	0.0	0.0
(5, 10]	0.2	0.0	0.0	0.0	0.0	0.0
> 10	0.1	0.0	0.0	0.0	0.0	0.0

6.3 Performance Test

The performance is another important factor to be considered in developing mathematical functions. It is measured in cycles for speed. To improve the resolution, multiple calling the math function is necessary.

6.3.1 Strategies for Performance Testing

Considering the implementation with CBE, the SPU decrementer is used to record the time elapsed in ticks for calling the math function with arguments in the working range. The implementation of our performance test includes two parts, the PPU code and SPU code.

The PPU code is responsible for

- Calculating every fixed value with a given working range and a mesh size
- Preparing an array of random arguments at each fixed value by calling the random function with a given array length
- Depending on the resolution, deciding if multiple calls are necessary
- Creating a SPU thread and waiting for it to finish
- Getting the time result at each fixed value and averaging the time

The flowchart of the PPU code is shown in Figure 6.3.
The job of the SPU code includes

- DMA transfer the array of random arguments into LS
- Recording the time elapsed for calling the math functions with SPU decrementer
- DMA transfer the time in ticks to MS

The flowchart of the SPU code is shown in Figure 6.4.

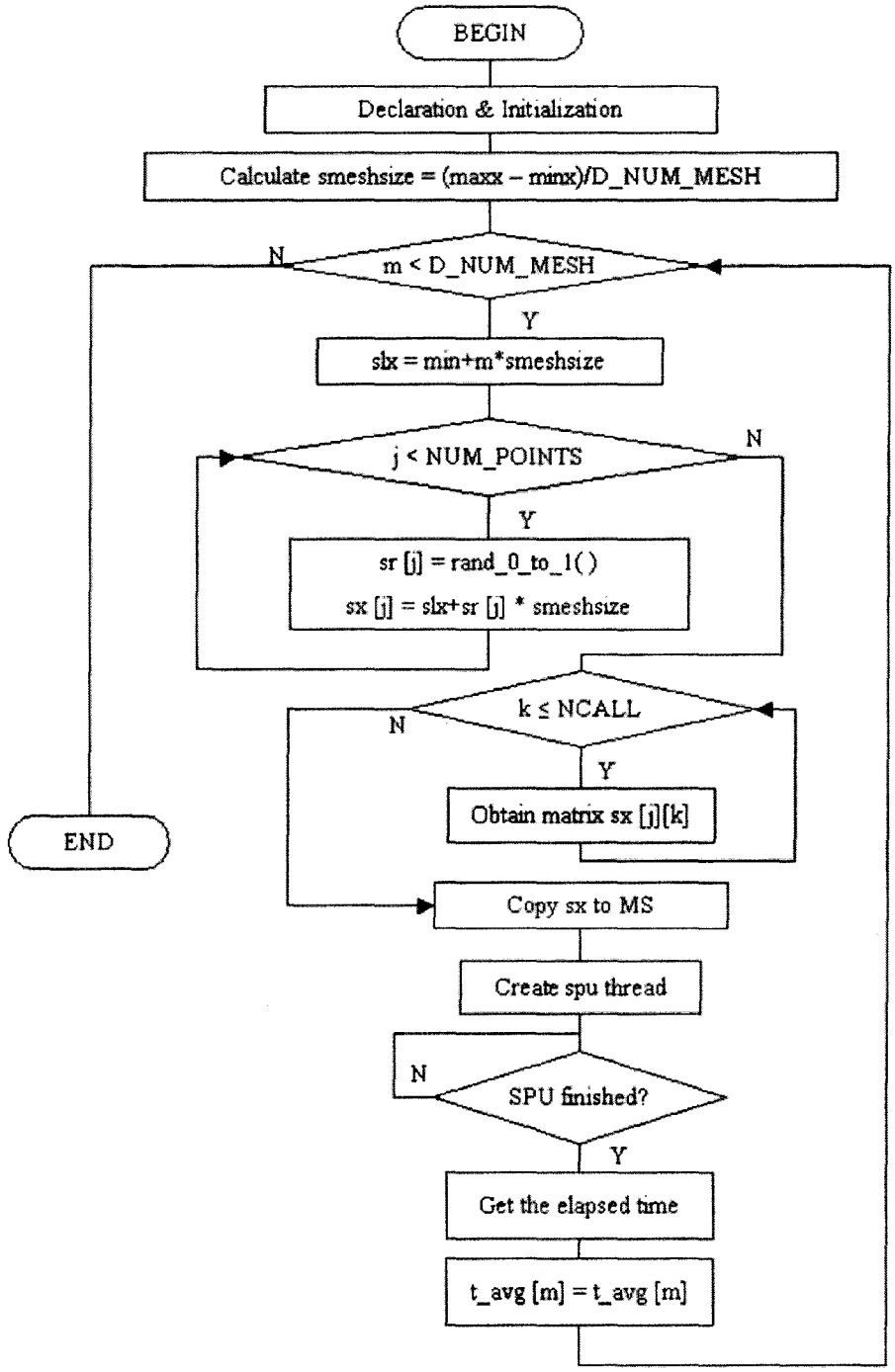

Figure 6.3: PPU Flowchart of Performance Test Implementation

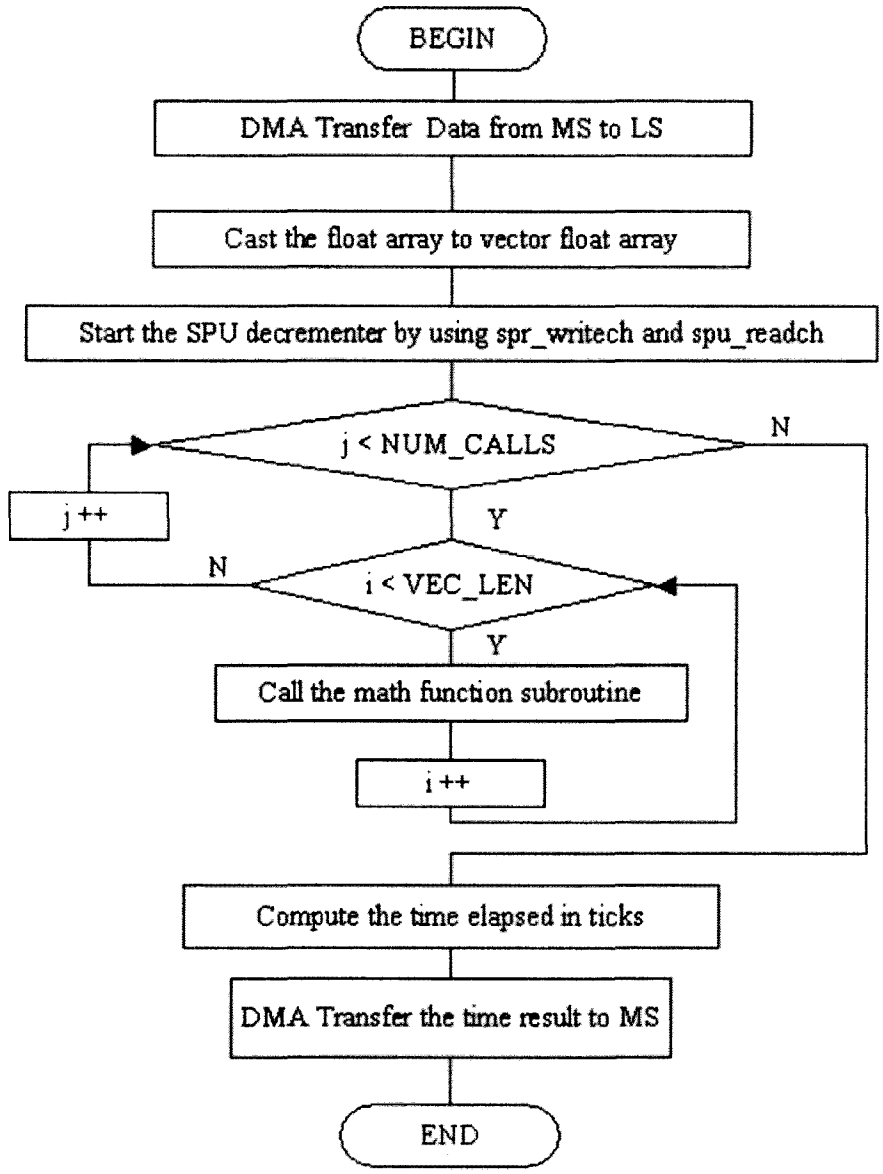

Figure 6.4: SPU Flowchart of Performance Test Implementation

6.3.2 Performance Test Results

The performance test is run on CBE processor. The five versions of each function include our library subroutine, the inline version, long vector with length of 1000 and the corresponding SDK1.1 library subroutine and inline function are all tested by running the same performance test code with different ways to call the same function. The results are displayed in Figures 6.5-6.10. In these figures, the horizontal axis represents the math functions with five versions and the vertical axis refers to the cycles per float taken to call the respective math version. Note that the output of the performance test is in unit of ticks. The ticks is then converted to cycles by a factor of 160.

It is easily found that the inline version wins over the library subroutine for all functions. The reason is that the calling inline function avoids calling overhead. It is also observed that most of our math versions (e.g., cbrt, exp and recip), lib subroutine and inline, have a better performance than the respective SDK ones. For the log functions (log, log10), their performance of library subroutine in SDK1.1 is a little bit better than ours, but the accuracy of our functions outperforms the SDK counterpart (see Table 6.3). And the version of long vector, which are not in SDK1.1, has the similar or a little better performance, compared to the corresponding inline function.

6.4 Summary

This chapter described the accuracy and performance tests for the recently developed single-precision SPU math functions for the CBE. Based on the architecture of the CBE and features of the floating-point format, a full range accuracy test and the performance test have been implemented for PPU and SPU as shown in flowcharts. The test results are displayed in plots and histograms.

The test results show the new developed single precision SPU functions outperform the SDK1.1 counterpart (when they exist) either in accuracy or performance or both.

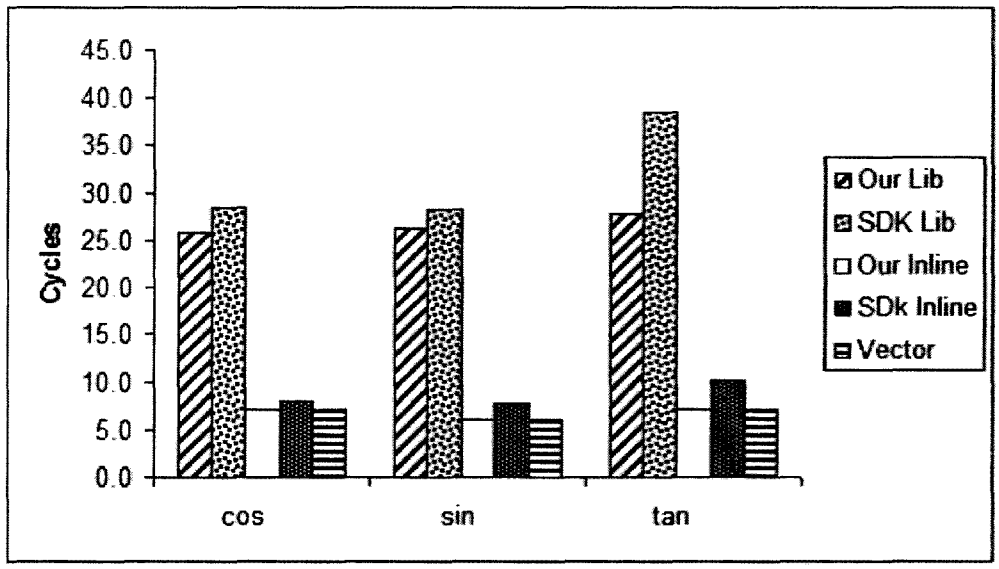

Figure 6.5: Performance Comparison of Trig Functions

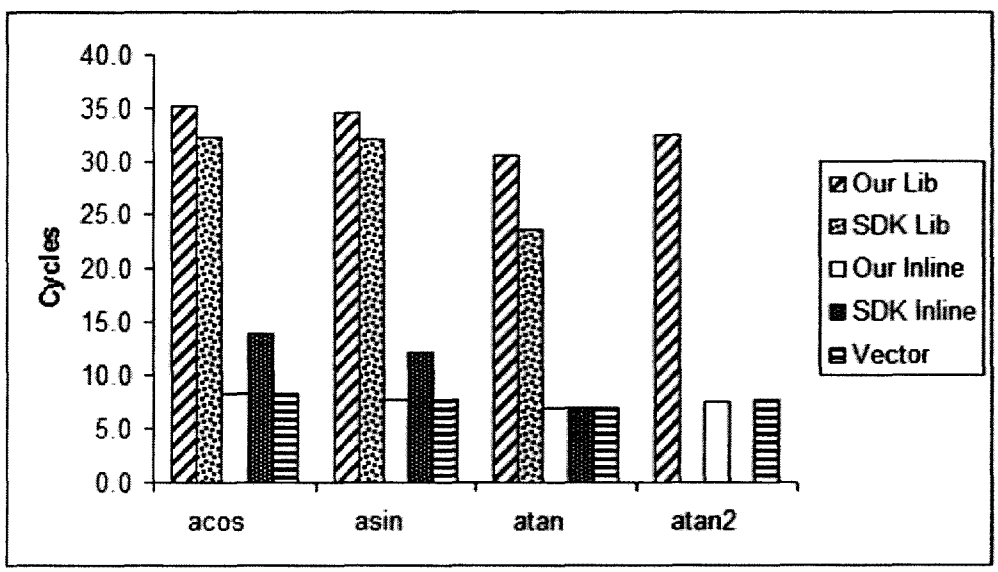

Figure 6.6: Performance Comparison of Inverse Trig Functions

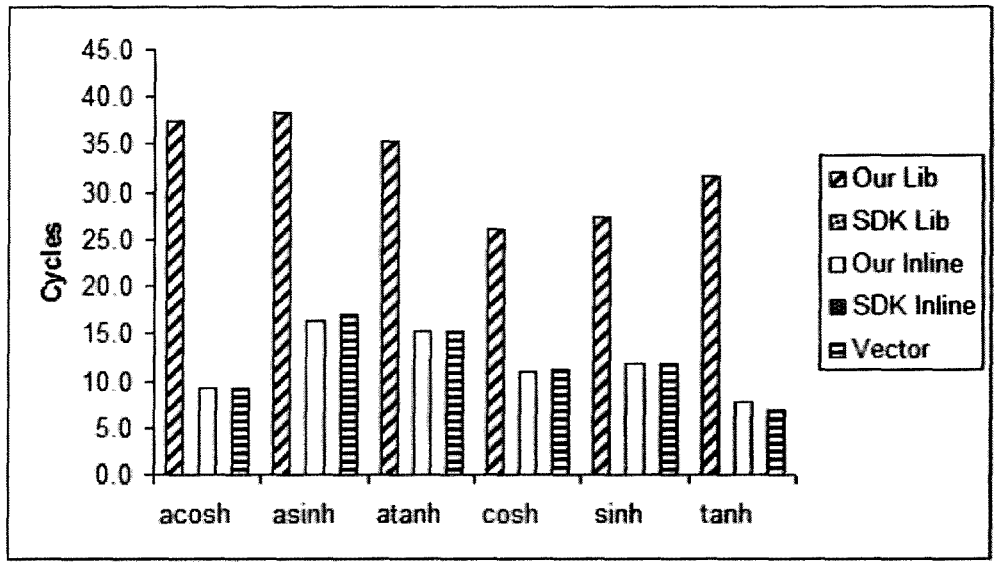

Figure 6.7: Performance Comparison of Hyperbolic Functions

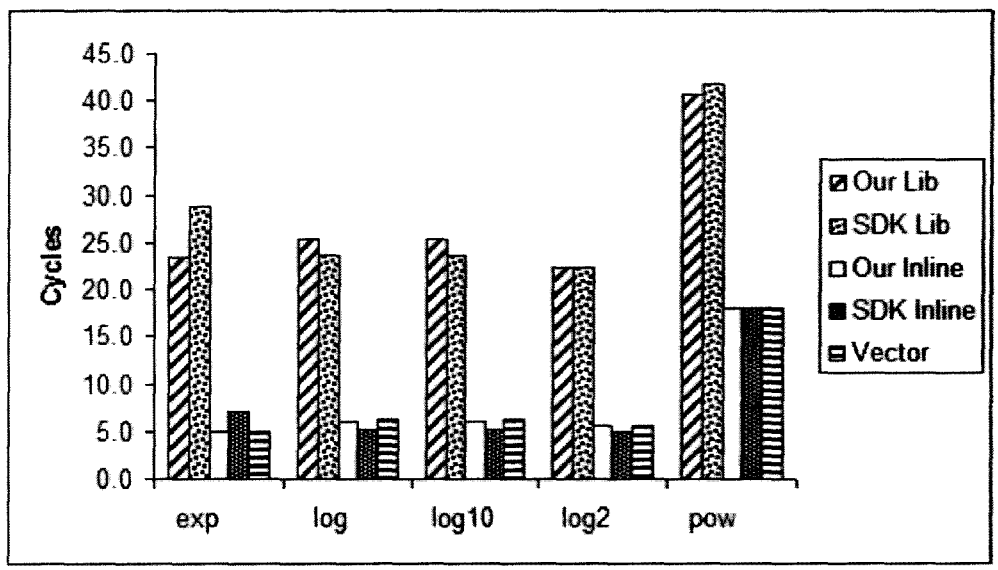

Figure 6.8: Performance Comparison of Exponents and Logarithms

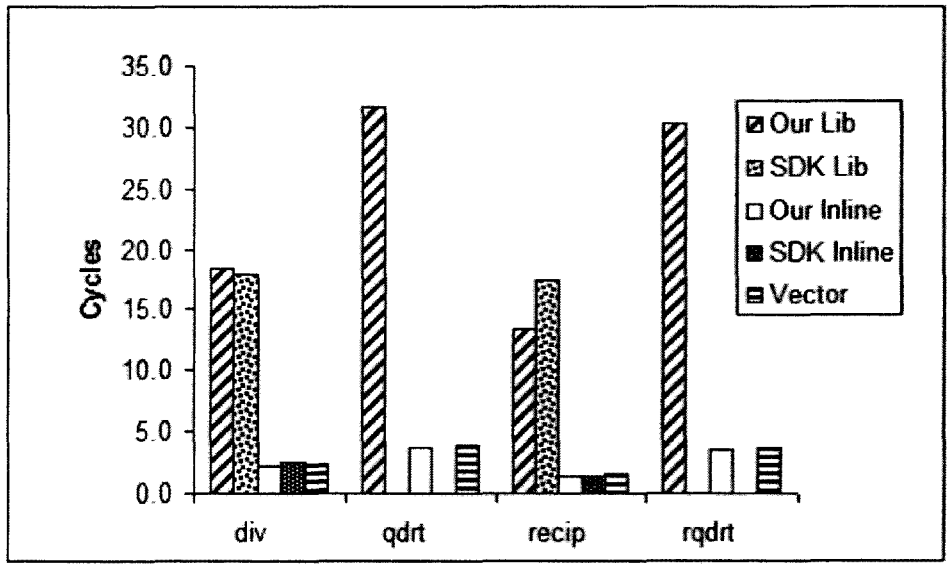

Figure 6.9: Performance Comparison of Division and Other Functions

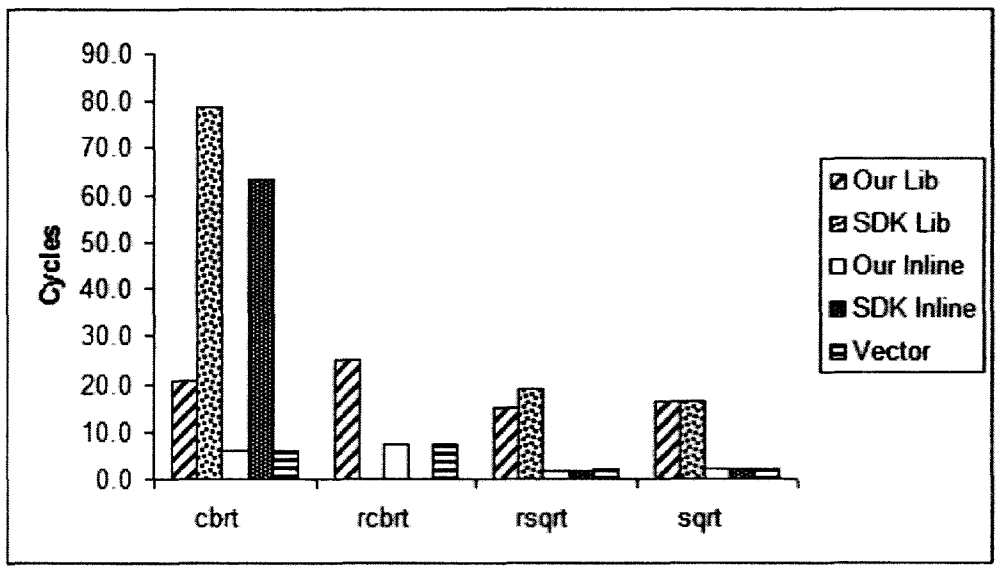

Figure 6.10: Performance Comparison of Square Root and Other Functions

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis comprises two parts. Part 1 is the algorithm development for MRI velocity quantification and its implementation on CBE. Part 2 is the evaluation of the SPU math functions on CBE. The math functions are called when implementing the algorithm on CBE.

The study on MRI velocity quantification from noisy images was presented in Chapter 3. Little work has been done in this research field, especially using an optimization technique. This work proposed in Chapter 3 is a new attempt in this field, that is, a mathematical way to formulate the problem.

A number of experiments have been done to verify the algorithm and to show the effects of the regularization parameter, magnetic field and noise. The results have shown that the regularization parameter affects the smoothness of the solution. If it is too big, the TV term will dominate the solution and the solution is too smooth and far from the real solution. If it is too small, the fitting term will dominate the solution, resulting high frequencies in the solution. From the experiments, we can conclude that there exists an optimal value of the regularization parameter, which is another topic in inverse problems and is not covered in this thesis.

The experiments have also demonstrate when environmental noise is stronger, the algorithm requires a stronger magnetic field. This is reasonable and can be realized

in practice. The correctness of the algorithm has been validated by numerical experiments. The estimated velocity converges to the real velocity very quickly.

The second part of this thesis is the practical implementation on CBE. The implementation of the MRI velocity quantification was described in Chapter 4. As we known, when implementing an algorithm on a particular hardware, the architecture of the hardware should be taken into account. This is an implementation issue and we explained why a modified algorithm was required, as presented in Chapter 4. The limitation on image size was derived. The experiments on CBE have proven that (i) the results perfectly matched the results from MATLAB, and (ii) the code on the SPU runs ten times faster than the same code on the PPU.

The implementation of accuracy and performance tests on CBE is the part of CAS project with IBM. Using ideas from an IBM test driver [34], the test code was developed for CBE SPU single precision math functions. The test results helped the developing team to tune the functions and provided evidence required for delivery and release. The math functions can be released with information about performance and accuracy, which will help developers make decision on choosing math function for their development.

7.2 Future Work

As we mentioned above, the research work involved in this thesis is a new attempt and the algorithm is only for one dimention. And the implementation method with CBE can be further improved. That leaves lots of work to be investigated and to be taken into account in the future. For example, main future work can be listed as below:

1. For simplicity, velocity change was considered along one dimensional only. Based on this work, the algorithm developed in Chapter 3 can be extended to the case with velocity change along two or three dimensional. The new Hessian Matrix will become complicated and solving the linear equations will be difficult too.

2. In the algorithm development, we ignored the part of total variation of density for simplicity. The TV term of density should be considered in the future.

3. Choosing the optimal value of the regularization parameter is also needed to be investigated.

4. One possible improvement for the algorithm implementation on CBE is to make use of the vector SPU intrinsics instead of scalar C commands. This will improve the performance.

5. The current algorithm uses only one SPU only. By making use of the power of data parallelism or function parallelism, the implementation can be spread over several SPUs, which is other way to improve the performance.

6. By using double DMA transfer and DMA list, the image size allowed to be processed will be larger.

7. The test code only works for single precision math functions. But it can be easily extended for double precision math functions.

All of above require solid background in MRI, numerical analysis, optimization and deep understanding of CBE and parallelism programming. Hopefully, this thesis provides a basis for solving the problem of MRI velocity quantification and for implementing the algorithm on real hardware.

The work associated with this thesis is also presented in three conferences, see Appendix A.

Appendix A–Conference List

1. Wei Li and Christopher Anand, MRI Velocity Quantification Using Iterative Method, poster session in *MOPTA06*, Waterloo, Canada, July 24 – 27, 2006

2. Wei Li, Christopher Anand and Robert Enenkel, Using Coconut to accelerate Vector MASS Functions for CELL, poster session in *Compiler Day of IBM CAS*, Toronto Lab, June 19, 2006

3. Wei Li, Christopher Anand and Robert Enenkel, MASS Test for CBE SPU, poster session in *CASCON06*, Toronto, Canada, Oct. 16 – 19, 2006

Appendix B–MATLAB Code

```
%-------------------------------------------------------------------------
%         File Name: vq1d.m
%         Uages: Iterative Algorithm for MRI VQ
%         Date: June 20, 2007
%-------------------------------------------------------------------------
clear all;
% Declare gloabel variable
e=1;         % the small positive number
Bx=0.01;     % without noise or noise only has real part
%Bx=1;       % when complex noise
NRa=11;      % Real noise amplifier
NIa=0;       % Imaginary noise amplifier
lamda=0.2;   % regularization parameter
M=16;        % # of rows of image
N=16;        % # of cols of image
n1=7;        % for true velocity
n2=10;
RUN=10;       % # of iterations
% read a image to get the density p
%---------------------
%p0=imread('mri_head16.bmp');
%p=im2double(p0);
%p=floor(256*im2double(p0));
%-----------------
```

```
p=128*ones(M,N);  % density
%[M N]=size(p);
%clear p0;
%figure(1);imshow(p);title('part of the original image');

% ---- Measured Image because of velocity ------
%V0=zeros(size(p));
V0=0.2*ones(size(p));  % True velocity
V0(:,n1:n2)=0.8;
Im=p.*exp(-i*Bx*V0);

% ---- Add gaussian noise -----------
%var=0.02;
%I=imnoise(I,'gaussian',0,var);
% add real noise
%In=randn(size(Im));
% Real noise if NIa=0
In=NRa*randn(size(Im))+i*NIa*randn(size(Im));
I=Im+In;

% Initial guess
Vx=0.1*ones(size(p));

% Iterations
for rp=1:RUN
%*********************************************************
%         the fitting term T1
% The first derivative of T1
T1VxJac=vecstack((2*p*Bx).*(real(I).*sin(Bx*Vx)+imag(I).*cos(Bx*Vx)));

% The second derivative of T1
T1VxD2=2*(Bx^2)*p.*(real(I).*cos(Bx*Vx)-imag(I).*sin(Bx*Vx));
```

```
T1VxD2s=vecstack(T1VxD2);
T1VxHes=zeros(M*N);
for j=1:M*N
    T1VxHes(j,j)=T1VxD2s(j);
end

% Solve the equation
%T1DeltaVxs=-(inv(T1VxHes))*T1VxJac;

% update the velocity matrix
%Vx=Vx+vecunstack(T1DeltaVxs,M);

%************************************************************
%         the penalty term T2
% The first derivative of T2
for k=1:M
    for j=1:N
        if j==1
           T2VxD1(k,j)=2*Vx(k,j)*(1/(2*(abs(Vx(k,j)))+e)+(1/(2*(abs...
           (Vx(k,j+1)-Vx(k,j)))+e)))-2*Vx(k,j+1)/(2*(abs(Vx(k,j+1)...
           -Vx(k,j)))+e);
        elseif j==N
           T2VxD1(k,j)=2*Vx(k,j)*(1/(2*(abs(Vx(k,j)-Vx(k,j-1)))+e)...
           +(1/(2*(abs(0-Vx(k,j)))+e)))-2*Vx(k,j-1)/(2*(abs(Vx(k,j)...
           -Vx(k,j-1)))+e);
        else
           T2VxD1(k,j)=2*Vx(k,j)*(1/(2*(abs(Vx(k,j)-Vx(k,j-1)))+e)...
           +(1/(2*(abs(Vx(k,j+1)-Vx(k,j)))+e)))-2*Vx(k,j-1)/(2*(abs...
           (Vx(k,j)-Vx(k,j-1)))+e)-2*Vx(k,j+1)/(2*(abs(Vx(k,j+1)...
           -Vx(k,j)))+e);
        end
    end
```

```
end

% Stack vector
T2VxJac=vecstack(T2VxD1);

%     The second derivative of T2
% all elements constitute the main diagonal of Hessian
T2VxD2Diag=zeros(M,N);
for k=1:M
    for j=1:N
        if j==1
           T2VxD2Diag(k,j)=2/(2*(abs(Vx(k,j)))+e)+2/(2*(abs...
           (Vx(k,j+1)-Vx(k,j)))+e);
        elseif j==N
           T2VxD2Diag(k,j)=2/(2*(abs(Vx(k,j)-Vx(k,j-1)))+e)+...
           2/(2*(abs(0-Vx(k,j)))+e);
        else
           T2VxD2Diag(k,j)=2/(2*(abs(Vx(k,j)-Vx(k,j-1)))+e)+...
           2/(2*(abs(Vx(k,j+1)-Vx(k,j)))+e);
        end
    end
end

% all elements above the main diagonal of Hessian
T2VxD2OffD=zeros(M,N-1);
for k=1:M
    for j=1:N-1
        if j==1
           T2VxD2OffD(k,j)=-2/(2*(abs(Vx(k,j)))+e);
        else
           T2VxD2OffD(k,j)=-2/(2*(abs(Vx(k,j)-Vx(k,j-1)))+e);
        end
```

```
    end
end

% Hessian of Term 2
T2VxHes=zeros(M*N);
for k=1:M
    for j=(k-1)*N+1:k*N
        if j==(k-1)*N+1
           T2VxHes(j,j)=T2VxD2Diag(k,j-(k-1)*N);
           T2VxHes(j,j+1)=T2VxD2OffD(k,j-(k-1)*N);
        elseif j==k*N
           T2VxHes(j,j)=T2VxD2Diag(k,j-(k-1)*N);
           T2VxHes(j,j-1)=T2VxD2OffD(k,j-(k-1)*N-1);
        else
           T2VxHes(j,j)=T2VxD2Diag(k,j-(k-1)*N);
           T2VxHes(j,j-1)=T2VxD2OffD(k,j-(k-1)*N-1);
           T2VxHes(j,j+1)=T2VxD2OffD(k,j-(k-1)*N);
        end
    end
end

% Combine the two terms
VxJac=T1VxJac+lamda*T2VxJac
VxHes=T1VxHes+lamda*T2VxHes

% Solve the equation
%DeltaVxs=-(inv(VxHes))*VxJac;

% recursive algorithm for solving linear equations
% Only work for 1D case
for k=2:1:M*N
    VxHes(k,k)=VxHes(k,k)-(VxHes(k-1,k)^2)/VxHes(k-1,k-1);
```

```
    VxJac(k)=VxJac(k)-(VxHes(k-1,k)*VxJac(k-1))/VxHes(k-1,k-1);
end

%for k=2:1:M*N
%    for j=2:M*N
%        if k==j
%           VxHes(k,j)=VxHes(k,j)-(VxHes(k-1,j)^2)/VxHes(k-1,j-1);
%        end
%    VxJac(k)=VxJac(k)-(VxHes(k-1,k)*VxJac(k-1))/VxHes(k-1,k-1);
%    end
%end

DeltaVxs(M*N)=VxJac(M*N)/VxHes(M*N,M*N)
for k=M*N-1:-1:1
    DeltaVxs(k)=(VxJac(k)-VxHes(k,k+1)*DeltaVxs(k+1))/VxHes(k,k);
end

% update the velocity matrix
Vx=Vx-vecunstack(DeltaVxs,M);

% norm 2 of the errors
MSE(rp)=norm(Vx-V0,2);%/norm(V0,2);

end
MSE

figure(1);
subplot(2,2,1),surfl(V0),axis([0 20 0 20 0 1]),
title('(a) True velocity');
subplot(2,2,2),surfl(Vx),axis([0 20 0 20 0 1]),
title('(b) Estimated velocity after 10 iterations');
subplot(2,2,3),plot(V0(2,:),'k-'),axis([0 20 0 1]);hold on,
```

```
subplot(2,2,3),plot(Vx(1,:),'k:'),plot(Vx(5,:),'k-.'),axis([0 20 0 1]),
title('(c) True (solid) and estimated (dotted)');
subplot(2,2,4),plot(MSE(:)),title('(d) Errors in L2'),
XLABEL('iterator'), axis([0 10 0 0.8]);

figure(2);
subplot(3,2,1),surf(abs(Im)),axis([0 20 0 20 100 150]),
title('(a) Amplitude of measured image');subplot(3,2,2),
surfl(angle(Im)),axis([0 20 0 20 -0.01 0]),
title('(b) Phase of measured image');
subplot(3,2,3),surfl(abs(In)),title('(c) Amplitude of noise');
subplot(3,2,4),surfl(angle(In)),title('(d) Phase of noise');
subplot(3,2,5),surfl(abs(I)),axis([0 20 0 20 100 150]),
title('(e) Amplitude of noisy image');
subplot(3,2,6),surfl(angle(I)),
axis([0 20 0 20 -0.01 0]),title('(f) Phase of noisy image');

%-------------------------------------------------------------------------
%         File Name: vecstack.m
%         Uages: convert a matrix to a vector
%         Date: June 20, 2007
%-------------------------------------------------------------------------
function v = vecstack(A)

% Stack vectors
[m n]=size(A);
v = zeros(m*n,1);
for i=1:m
v((i-1)*n+1:i*n)=A(i,:);
end

%-------------------------------------------------------------------------
```

```
%          File Name: vecunstack.m
%          Uages: convert a vector to a matrix
%          Date: June 20, 2007
%-------------------------------------------------------------------------
function A = vecunstack(v,m)

% Unstack vectors
mn=length(v);
n=mn/m;
A=zeros(m,n);
%v=v';
for i=1:m
    A(i,:)=v((i-1)*n+1:i*n);
end
```

Bibliography

[1] E. Mark Haacke, Robert W. Brown, Michael R. Thompson, Ramesh Venkatesan. *Magnetic Resonance Imaging– Physical Principles and Sequence Design.*, John Wiley & Sons,1999.

[2] Christopher Anand, *Notes of Course–Regularized Image Reconstruction*, May 2006.

[3] Dwight G. Nishimura, *Principles Of Magnetic Resonance Imaging*, April 1996.

[4] Tony F. Chan, Jianhong Shen. *Image processing and Analysis.*, SIAM, 2005

[5] IBM. *CBE Broadband Engine Handbook*, May 2006, Version 1.0.

[6] Michael Perrone, *Introduction to the CBE Processor–Lecture 2*, http://cag.csail.mit.edu/ps3/lectures/6.189-lecture2-cell.pdf, IBM, 2007.

[7] Nicholas Blachford, *CBE Architecture Explained*, 2005, Version 2

[8] M. Hanke and P.C. Hansen, "Regularization methods for large-scale problems," *Survey on Mathematics for Industry*, Vol.3 Issue (4), pp. 253–315, 1993.

[9] Oraintara, S. Karl, W.C. Castanon, D.A. Nguyen, T.Q., "A method for choosing the regularization parameter in generalized Tikhonov regularized linear inverse problems," *2000 International Conference on Image Processing*, Vol. 1, pp. 93–96, Sep. 2000.

[10] Murat Belge, Misha E Kilmer and Eric L Miller, "Efficient determination of multiple regularization parameters in a generalized L-curve framework," *Electronic Journals on Inverse Problems*, pp. 1161–1183, July 2002.

[11] C.R.Vogel, "Non-Convergence of the L-curve Regularization Parameter Selection Method," *Inverse problems*, Vol.12, pp. 535–547, 1996.

[12] Jesper Pedersen, "Modular Algorithms for Large-Scale Total Variation Image Deblurring," *IMM-Thesis-2005-06*, 2005.

[13] Vivek Agarwal, "Total Variation Regularization and L-curve method for the selection of regularization parameter," *Notes on ECE 599*, summer 2003.

[14] T. F. Chan, G. H. Golub, and P. Mulet, "A nonlinear primal–dual method for total variation–based image restoration," *SIAM* Vol. 20, No. 6, pp. 1964–1977, 1999.

[15] David Tschumperle and Rachid Deriche, "Variational Frameworks for DT-MRI Estimation, Regularization and Visualization," *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference*, Vol1.1, pp. 116–121, Oct. 2003.

[16] David Tschumperle and Rachid Deriche, "Vector-Valued Image Regularization with PDEs: A Common Framework for Different Applications," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, Vol. 27, No. 4, pp. 506–517, April 2005.

[17] David C.Dobson and Fadil Santosa, "Recovery of Blocky Images from Noisy and Blurred Data", *SIAM Journal on Applied Mathematics*, Vol. 56, Issue 4, pp. 1181–1198, Aug 1996.

[18] Curtis R.Vogel, "A fast, robust algorithm for total variation based reconstruction of noisy, blurred images", *IEEE Transaction of Image Processing*, Vol.7, pp. 813–824, July 1998.

[19] Peter Blomgren, Tony F.Chan, Pep Mulet and C.K. Wong, "Total Variation Image Restoration: Numerical methods and Extension," *IEEE International Conference on Image Processing*, Vol. III, pp. 384–387, 1997.

[20] Jeffrey A. Fessler and Scott D. Booth, "Conjugate-Gradient Preconditioning Methods for Shift-Variant PET Image Reconstruction," *IEEE Trans. on Image Processing*, Vol. 8, N0. 5, pp. 688–700, May 1999.

[21] E. Artzy, T. Elfving and G. T. Herman, "Quadratic optimization for image reconstruction, II," *Comput. Graph. Image Processing*, Vol. 11, pp. 242–261, 1979.

[22] S. Kawata and O. Nalcioglu, "Constrained iterative reconstruction by the conjugate gradient method," *IEEE Trans on Medical Image*, Vol. 4, pp. 65–71, June 1985.

[23] J. G. Nagy, R. J. Plemmons, and T. C. Torgersen, "Iterative image restoration using approximate inverse preconditioning," *IEEE Trans. on Image Processing*, Vol. 5, pp. 1151–1162, July 1996.

[24] N. H. Clinthorne et al., "Preconditioning methods for improved convergence rates in iterative reconstructions," *IEEE Trans. Med. Imag.*, Vol. 12, pp. 78–83, Mar. 1993.

[25] Tony F. Chan, H.M. Zhou and Raymond H. Chan, "Continuation Method for Total Variation Denoising Problems," *SPIE Symposium for Advanced Signal Processing*, Vol. 25, 1995.

[26] R. Fletcher. *Practical Methods of Optimization.* Wiley, 1987. Second Edition.

[27] P. E. Frandsen, K. Jonasson, H. B. Nielsen, and O. Tingleff, *Unconstrained optimization*, 2004, 3rd edition.

[28] J. C. DiCarlo, K. S. Nayak, B. S. Hut, D. G. Nishimura, and J. M. Pauly, "Cardiac-Gated Multi-Shot Fourier Velocity-Encoding." *Proceedings ISMRM Tenth Scientific Sessions*, pp. 1802, May 2002.

[29] Julie C. DiCarlo, Brian A. Hargreaves, Krishna S. Nayak, Bob S. Hu, John M. Pauly, and Dwight G. Nishimura, "Variable-Density One-Shot Fourier Velocity Encoding," *Magnetic Resonance in Medicine*, No. 54, pp. 645–655, 2005

[30] Stephen R. Schach, *Software Engineering with JAVA*, Fourth Edition, McGraw-Hill, March 1999.

[31] David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, No. 1, pp. 5–48, March 1991

[32] SUN microsystems, *Numerical Computation Guide*, e-book, http://docs.sun.com/source/806-3568/ncgTOC.html.

[33] Steve Hollasch, *IEEE Standard 754 Floating Point Numbers*, http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

[34] Robert Enenkel, "A Comprehensive Test Environment for Mathematical Functions," *IBM Technical Report* TR-74.200, 2004.

[35] S. Gal and B. Bachelis, "An Accurate Elementary Mathematical Library for the IEEE Floating Point Standard," *ACM TOMS*, Vol. 17, No. 1, pp. 26–45, Mar. 1991.

[36] W. J. Cody, "Performance Evaluation of Programs for the Error and Complementary Error Functions," *ACM TOMS*, Vol. 16, No. 1, pp. 29–37, Mar. 1990.