Fault Location Using Wide-Area Measurements and Sparse Estimation

A Thesis Presented

by

Guangyu Feng

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Electrical and Computer Engineering

Northeastern University
Boston, Massachusetts

November 2015
NORTHEASTERN UNIVERSITY
Graduate School of Engineering
Thesis Signature Page

Thesis Title: Fault Location Using Wide-Area Measurements and Sparse Estimation
Author: Guangyu Feng  NUID: 001916249
Department: Electrical and Computer Engineering

Approved for Thesis Requirements of the Master of Science Degree

Thesis Advisor
Dr. Ali Abur
Signature Date

Thesis Committee Member or Reader
Dr. Hanoch Lev-Ari
Signature Date

Thesis Committee Member or Reader
Dr. Bahram Shafai
Signature Date

Department Chair
Dr. Sheila S. Hemami
Signature Date

Associate Dean of Graduate School:
Dr. Sara Wadia-Fascetti
Signature Date
To my parents.
Contents

List of Figures v
List of Tables vi
List of Acronyms vii
Acknowledgments viii
Abstract of the Thesis ix

1 Introduction 1
1.1 Motivation of the Study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Contribution of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

2 Background 4
2.1 State of Art of Fault Location Methods . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.1 Traveling Wave-based Methods . . . . . . . . . . . . . . . . . . . . . . . . 4
2.1.2 Impedance-based Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.3 AI-based Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Technical Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.1 Wide Area Measurement System . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.2 Sparse Estimation Techniques . . . . . . . . . . . . . . . . . . . . . . . . . 6

3 Basic Methodology 7
3.1 Transformation of Fault Location Problem . . . . . . . . . . . . . . . . . . . . . . . 7
3.1.1 Derivation of Equivalent Injection . . . . . . . . . . . . . . . . . . . . . . . . 7
3.1.2 Determination of Fault Type and Fault Line Type . . . . . . . . . . . . . . . . . 11
3.1.3 Simulation Verification of Equivalent Injection Theory . . . . . . . . . . . . . . . . 12
3.1.4 Building the Linear Estimation Equations . . . . . . . . . . . . . . . . . . . . . . 13
3.2 Sparse Estimation Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
3.3 Measurement Placement for Unique Solution . . . . . . . . . . . . . . . . . . . . . . 16
# Alternations Improving the Method

4.1 Overlapping Group Lasso .................................................. 19
4.2 Second Stage OLS Estimation .............................................. 20
   4.2.1 Criteria for Valid Solution .......................................... 20
   4.2.2 OLS Backup Estimation Routine ................................... 20
4.3 Extended Robust Formulation and Correction Routine .................. 21
   4.3.1 Extended Robust Formulation ...................................... 21
   4.3.2 Recursive Solution Routine with Correction of Erroneous Measurement(s) 21
4.4 Parallel Implementation .................................................... 22
   4.4.1 Problem Decomposition ............................................. 22
   4.4.2 Parallelism Attempted .............................................. 24

# Simulation Results

5.1 Faults in the Unbalanced Test Case ..................................... 28
5.2 Wide-area Cases ............................................................. 29
   5.2.1 Teed Circuit Fault ................................................... 29
   5.2.2 Comparison of Basic LARS-Lasso and Overlapping Group Lasso .... 30
   5.2.3 Robustness of Extended LARS-Lasso Formulation Against Gross Error 33
5.3 Parallel Implementation Results .......................................... 34
   5.3.1 Sequential Run Time Benchmark .................................. 34
   5.3.2 Speed-up Using openMP ............................................ 34
   5.3.3 Speed-up Using GPU ............................................... 35

# Conclusion

6 Conclusion ............................................................................. 37

# Bibliography

Bibliography ............................................................................. 39

# Appendix

A Appendix .............................................................................. 45
   A.1 Matlab Code Solving Lasso Problem .................................. 46
   A.2 Parallel C Code using openMP ....................................... 51
   A.3 CUDA C Code for GPU implementation ........................... 81
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Illustration of a fault on a two terminal line.</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Illustration of a fault on a two terminal line with parallel branch.</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Illustration of a fault on teed line.</td>
<td>10</td>
</tr>
<tr>
<td>3.4</td>
<td>Identification of fault type and faulted line type through inspecting equivalent injections.</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>11-bus distribution system diagram in ATP-EMTP.</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>Real part of equivalent injections for the 11-bus unbalanced case.</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>Imaginary part of equivalent injections for the 11-bus unbalanced case.</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Flowchart for the recursive estimation.</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Three Scenarios of Data Distribution.</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Example of Column Block Distribution.</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>Development process of different implementations.</td>
<td>25</td>
</tr>
<tr>
<td>5.1</td>
<td>Computation Times Used by Basic LARS-Lasso and Overlapping Group Lasso.</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Number of Iterations by Basic LARS-Lasso and Overlapping Group Lasso.</td>
<td>31</td>
</tr>
<tr>
<td>5.3</td>
<td>Error of Fault Distance by Basic LARS-Lasso and Overlapping Group Lasso.</td>
<td>32</td>
</tr>
<tr>
<td>5.4</td>
<td>Number of OLS Backup Estimations by Basic LARS-Lasso and Overlapping Group Lasso.</td>
<td>32</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of Fault Distance Error under One or Two Unit Failures.</td>
<td>33</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparison of Speed-ups for Different Cases using openMP.</td>
<td>35</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Relative Error of Fault Distance for the Unbalanced 11-Bus Case</td>
<td>28</td>
</tr>
<tr>
<td>5.2</td>
<td>Platform Information</td>
<td>34</td>
</tr>
<tr>
<td>5.3</td>
<td>Sequential Run Times</td>
<td>34</td>
</tr>
<tr>
<td>5.4</td>
<td>Summary of GPU run times</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of Profiling Results</td>
<td>36</td>
</tr>
</tbody>
</table>
List of Acronyms

AI  Artificial Intelligence
WAMS  Wide Area Measurement System.
PMU  Phasor Measurement Unit.
Lasso  Least Absolute Shrinkage and Selection Operator.
The Lasso is a shrinkage and selection method for linear regression. It minimizes the usual sum of squared errors, with a bound on the sum of the absolute values of the coefficients. It has connections to soft-thresholding of wavelet coefficients, forward stagewise regression, and boosting methods.
LARS  Least Angle Regression. The “S” suggests “Lasso” and “Stagewise”.
OLS  Ordinary Least Squares.
openMP  Open Multi-Processing
MPI  Message Passing Interface
GPU  Graphics Processing Unit
CUDA  Compute Unified Device Architecture
Acknowledgments

Here I wish to thank those who have supported me during and before the process of the thesis work. Among them, I’m especially grateful to Dr. Ali Abur, who has shared with me valuable research guidance, wisdom as well as unique insights about problem-solving and debugging. I always consider it a privilege to have taken the power system analysis and unbalanced power grid course offered by Dr. Abur since these courses not only made my knowledge on power system more comprehensive but also left me with notes and codes that are still of great help to my research. In addition, I appreciate it very much the chances that Dr. Abur gave me for publishing and presenting my work even when I was feeling not very confident about my results.

I would like in the mean time thank Dr. Lev-Ari and Dr. Shafai in my committee as well as Dr. Tomsovic from Univ. of Tennessee and Dr. Leeser from whom I have taken important courses on signal processing, linear system and control, constrained optimization and high performance computing. Beside enhancing my basic engineering skills, all these courses have contributed directly or indirectly to my research.

Furthermore, I really appreciate the encouragement and help from my lab mates (or former lab mates) especially Mert, Murat, Alireza, Chenxi, Pengxiang and Yuzhang. I wish all the best to their life and careers.

Last but not least, I can not thank my parents more for supporting me remotely both financially and spiritually over the past years. They are always unconditionally standing behind me no matter what choice I make.
Abstract of the Thesis

Fault Location Using Wide-Area Measurements and Sparse Estimation

by

Guangyu Feng

Master of Science in Power System

Northeastern University, November 2015

Dr. Ali Abur, Adviser

This thesis describes a fault location method which relies on scattered wide-area synchronized phasor measurements and usage of sparse estimation techniques. The main contribution is the way fault location is reformulated as a sparse estimation problem. Faulted system is modeled by equivalent terminal bus injections which would cause the same changes in bus voltages with the fault current drawn at any point along the faulted line. Once these injections are estimated, the fact that the ratio between equivalent injections at the two terminal buses depends only on the ratio of serial impedances on each side of the fault point can be used to locate the fault. It is shown that this formulation applies to both two terminal lines as well as teed lines regardless of fault type or resistance. Assuming availability of an accurate three-phase network model and a sufficient number of phasor measurements over the entire network, an underdetermined set of linear equations can be formed and then solved for the sparse equivalent bus injections. Then the problem fits naturally into a Least Absolute Shrinkage and Selection Operator (Lasso) formulation and can be solved via the Least Angle Regression (LARS) algorithm. Based on the condition for unique solution for Lasso problem, a scheme for optimal phasor measurement placement is also derived. Furthermore, alternations have been made to the basic implementation of LARS so that the method’s reliability, robustness and efficiency is improved.

Using extensive simulations on both unbalanced three phase test case and wide-area network test case, the accuracy and efficiency of the proposed method with its alternations have been verified.
Chapter 1

Introduction

Power transmission lines are exposed to faults caused by natural or artificial reasons such as lightening, short circuits, faulty equipments, mis-operation, overload and aging. Faults can be roughly categorized as open circuit faults and short circuit faults. While open circuit faults are generally rare, short circuit faults are more common in power systems and attract wide-spread attention. Faults are also divided by being temporary or permanent. Temporary faults are known to be able to self-clear and are more common on power lines. For these faults, accurate location of the fault can find the weak point on the line and prevent future occurrence of fault by reparation. Permanent faults, on the other hands, cause major damages and require immediate action from the crew to make the faulted line return to service. Untimely or inaccurate fault location can result in prolonged line outage and also severe economic losses and reduced reliability of the service. Thus accurate and timely fault location has always been a critical issue in power system operation [1].

When a fault occurs on a transmission line, distance relays can provide some indication of the general area where the fault is, but they are designed more in a way that line protection is made on-line as fast as possible other than pinpoint the fault location. On the other hand, distance to fault is estimated off-line from the recorded data. Depending on the types of data and models available to researchers, there have been various methods proposed so far with most of them relying on local data from one or both ends of the faulted transmission line.
CHAPTER 1. INTRODUCTION

1.1 Motivation of the Study

While there have been well-established fault locators using local measurements, chances are that local measurement units may fail at the very time when the fault occurs. In this regard, this work has turned to wide-area measurements for more reliable fault location method. Recent years have seen increasing deployment of synchronized phasor measurements in power system and applications based on the Wide Area Measurement System (WAMS) such as wide-area control and linear state estimation. But the benefit of WAMS on fault location is not yet fully discovered since power grid protection system have long been designed with local measurements as the decision variables.

Furthermore, previous formulations of fault location problems often need prior information about fault type or faulted line to determine how the fault can be located. This adds to the purpose of this work to derive a fault location method of universal applicability and straightforward solution regardless of fault types and faulted line types.

In formulating the proposed fault location method, the advancement of sparse estimation technique is also taken into account so that fault location problem can be tackled using new mathematical tools, contributing to the diversity of the pool of fault locators.

1.2 Contribution of the Thesis

This work contributes the current fault location practice in the following aspects:

- Non-local, in other words, wide-area phasor measurements is used to locate the fault, thus making it still possible to locate the fault when local measurement fails or is broken down.

- Transformation of fault location problem into linear sparse estimation problem of general form regardless of fault types and faulted line types, enabling the usage of sparse estimation algorithms.

- Only partial observability of the system is required to ensure unique fault location solution.
CHAPTER 1. INTRODUCTION

• Alternations to basic sparse estimation implementation are made to improve the reliability and robustness of the method.

• In addition to sequential implementation, parallel implementations of the proposed method are also investigated and realized, improving the method’s efficiency.

1.3 Thesis Outline

This thesis is composed of six chapters, the second chapter reviews past literatures on fault location and provides technical background for the proposed method. The third chapters introduces the formulation and basic proposed methodology. Then chapter 4 presents several alternations made on the basic method so as to obtain better performance in terms of accuracy, robustness and efficiency. Chapter 5 describes the test result addressing the universal applicability and performance of the proposed method. Finally chapter 6 concludes the thesis and introduces future work.
Chapter 2

Background

2.1 State of Art of Fault Location Methods

Fault location methods currently used for transmission system faults can be broadly classified as (1) traveling wave-based methods, (2) impedance-based methods and (3) Artificial Intelligence (AI)-based methods.

2.1.1 Traveling Wave-based Methods

Traveling-wave based methods [2–6] make use of the high frequency (several kHz to MHz) electromagnetic transients induced by the fault to locate the time of arrival of the traveling wave from the fault point. Thus, modal domain wave velocities need to be calculated in order to obtain the distance from the fault point to the measuring device where the traveling wave is recorded. While thanks to high frequency sampling, traveling wave based methods are independent with fault types, the wave velocities can not always be accurately defined given the variation of the velocities with frequency and tower configurations. In addition, there usually is a non-negligible cost associated with the required high-frequency sampling of the waveforms.
CHAPTER 2. BACKGROUND

2.1.2 Impedance-based Methods

Impedance based methods are methods that use phasors to locate the fault. There is a large volume of papers describing such methods using single-ended or double-ended measurements (e.g. [7–10]). These methods require at least one of the terminals of the faulted line to be equipped with a monitoring device like a digital fault recorder or other types of IEDs (intelligent electronic devices) [11]. There are also multi-end algorithms [9,12–17] that use measurements from the multiple ends of transmission line. For algorithms requiring more than one end of measurements, they can be further divided into ones using synchronized or unsynchronized measurements. As stated in Sec. 1.1, many impedance methods require prior assumption or knowledge about the fault section as well as fault type to determine the detail of their algorithm, making them uneasy to implement.

2.1.3 AI-based Methods

AI-based methods [18–22] are those using artificial intelligence techniques such as pattern recognition and machine learning algorithms. Unlike impedance or traveling wave based methods, they do not rely on physics relationships between the measurement and the fault location but rather look for the underlying connection between certain feature of the data and the fault. As the configuration of the system evolves, these methods often need to be “retrained” to better adjust to the system.

2.2 Technical Background

2.2.1 Wide Area Measurement System

Recent years have seen major changes in power system operation due to the employment of WAMS. Many applications, mostly regarding monitoring and control actions have been improved. Yet it has not been clear how power system protection or fault location can benefit from this technical advantage.

Recently several studies have proposed the use of wide-area phasor measurements and locating the fault not just based on single or double-ended measurements, but based on several measurements taken at various locations over the whole system. The work in [23] derived a fault location factor as
CHAPTER 2. BACKGROUND

a function of fault distance for every bus. The homogeneity of this factor over a network is used to identify firstly the fault region and then the exact location. In [24], the expected change of phasors due to faults at different points of a network are used to be matched with measured phasors using an optimization scheme. Similar to [23], its diagnosis process is also hierarchal.

2.2.2 Sparse Estimation Techniques

Another important technical basis for this work is the use of sparse estimation techniques in power systems. In fact this kind of techniques have been used by various researchers for purposes of detection and identification of faults. One recent work reported in [25] uses several compressive sensing algorithms to pinpoint the location of a faulted bus. In [26] the authors employ greedy orthogonal matching pursuit (OMP) method and the least-absolute shrinkage and selection operator (Lasso) to identify line outages at affordable complexity. In another study [27] the authors derive an algorithm to detect and localize malicious data attacks using $L_1$ regularization for the generalized likelihood ratio test.
Chapter 3

Basic Methodology

3.1 Transformation of Fault Location Problem

3.1.1 Derivation of Equivalent Injection

The proposed method aims to use the pre-fault steady state model of a power system and represent a fault or multiple faults of any type on any combination of phases by current injection into the fault point so as to keep the consistency of the system model. To discover how a fault can be identified using the pre-fault system model, a series of classical fault analysis is conducted. Consider a single phase to ground fault in an $N$ bus system, first an extended bus impedance matrix $\tilde{Z}_{bus}$ is built with the fault point as the $(N+1)th$ node. The change of bus voltages due to the fault is computed by multiplying the $(N+1)th$ column of $\tilde{Z}_{bus}$ with a fault current $I_f$. Then using (3.1) where $Y_{bus}$ and $V$ represent the bus admittance matrix and bus voltages of the original $N$ bus system, a sparse $\Delta I$ is derived and the sparse bus current injections can be regarded as equivalent with the fault current injection in terms of their effect on bus voltages.

$$Y_{bus} \Delta V = \Delta I$$  \hspace{1cm} (3.1)

The result unveils an interesting feature that the nonzero entries in $\Delta I$ only exist in the terminal
buses of the faulted line and their ratio is determined solely by the serial impedances of branches
incident to the fault point. If the faulted line is homogeneous, the distance from the fault point to
each terminal can be readily calculated from the ratio of the equivalent injections which is a real
number. Otherwise it would take some effort to look up the line parameters for translating the
ratio of injections to ratio of distances, which is not within the scope of this paper. Note that the
aforementioned feature does not require the system to be balanced, nor does it require the shunt
capacitances to be neglected, the only pre-requisition is accurate Pi-models of lines and load data at
the time of the fault. Since the bus impedance matrix for any system during the fault instant can be
seen as a constant matrix, the rule of superposition can be applied for multi-phase faults and multiple
faults in the system, their equivalent bus injections will appear in multiple phases or multiple terminal
buses instead of a single pair in the case of single phase to ground fault. It has to be clarified that
although the equivalent injection representations of fault have been built from scratch by the authors,
it also appears in literatures such as [28] and has been utilized in a different formulation.

To support the general applicability of the proposed method, detailed demonstration of the
equivalent injections will be introduced separately for two and three terminal lines.

3.1.1.1 Two terminal lines

For conventional two terminal lines, consider the scenario shown in Fig. 3.1 where the terminal
buses are indexed as $A$ and $B$ and the serial line impedances from the fault point to the terminals are
named $z_1$ and $z_2$ respectively. Here the single phase representation is used for the sake of simplicity.

Following the derivation in Sec. 3.1.1 the injections equivalent to the fault current $I_f$ are:
CHAPTER 3. BASIC METHODOLOGY

As shown in (3.4) the ratio of the derived equivalent injections is inversely proportional to the ratio of the serial impedance on each side of the fault point and has nothing to do with the rest of the network.
3.1.1.2 Teed lines

Representation of faults by equivalent bus injections applies not only to conventional two terminal lines, but also to three terminal lines where the T-node is typically not monitored. In Fig. 3.3 the terminal buses are named as A, B and C and the T-node as T. \( z_1 \) and \( z_2 \) are the impedances between the fault point and bus A and the T-node respectively. The branch impedances of line \( B - T \) and \( C - T \) are denoted by \( z_{BT} \) and \( z_{CT} \).

![Diagram of a fault on teed line.](image)

Given a fault between A and T with a fault current \( I_f \), the equivalent terminal injections will be given by:

\[
I_A = I_f \left( 1 - \frac{z_1 (z_{BT} + z_{CT})}{z_{AT} z_{BT} + z_{AT} z_{CT} + z_{BT} z_{CT}} \right) \tag{3.5}
\]

\[
I_B = I_f \frac{z_1 z_{CT}}{z_{AT} z_{BT} + z_{AT} z_{CT} + z_{BT} z_{CT}} \tag{3.6}
\]

\[
I_C = I_f \frac{z_1 z_{BT}}{z_{AT} z_{BT} + z_{AT} z_{CT} + z_{BT} z_{CT}} \tag{3.7}
\]

\[
\frac{I_B}{I_C} = \frac{z_{CT}}{z_{BT}} \tag{3.8}
\]

Although the relationships between the injections seem more complicated than in (3.4), they are
still determined only by the ratios of the serial branch impedances on the faulted line. By comparing \( I_B \) and \( I_C \), the line segment \( A - T \) can be readily identified as the faulted segment and the distance to fault point can then be determined by either comparing \( I_A \) and \( I_B \) or \( I_A \) and \( I_C \).

### 3.1.2 Determination of Fault Type and Fault Line Type

From the derivations in 3.1.1 it can be concluded that the number and location of nodes with equivalent injection inherently indicates the type of the faulted line. When multi-phase fault occurs, the pair of equivalent injection on nodes of the same phases would still present the proportional relationships in (3.4) and (3.8). The only difference is that the sparsity of the computed current injection vector in two and three phase fault cases will be respectively two and three times of the sparsity in single phase fault case. Based on the Kirchhoff’s Law, if all elements of the equivalent injections sum up to zero, it indicates that the fault is ungrounded, otherwise it is a grounded fault. Therefore there will be no need to know the type of the faulted line and the fault or adjust the approach before computing these equivalent injections. Instead, they can be determined by simple inspection following the decision making diagram below.

![Decision Making Diagram](image)

**Figure 3.4:** Identification of fault type and faulted line type through inspecting equivalent injections.
CHAPTER 3. BASIC METHODOLOGY

Since the fault branch of any type is replaced by current source branch, the feasibility of the equivalent injection representation and decision making diagram is not affected by the value of fault resistance as long as it is not so large that the fault current can be neglected. The proposed method is therefore independent of fault resistance, which is not commonly seen in the field of fault location.

3.1.3 Simulation Verification of Equivalent Injection Theory

To add the credibility of the equivalent injection theory, simulation have been run in ATP-EMTP regarding a fault in an unbalanced system (Fig. 3.5) modified from the 13-Bus feeder case in [29]. All 11 buses and 10 line sections are marked on the picture. A phase A and B to ground fault on the 1/4 part of section 5 between Bus 2 and 6 has been simulated. Using the simulated change of bus voltages, the equivalent current injections have been computed. Fig. 3.6 and Fig. 3.7 are plots of the real and imaginary part of the current injection vector.

Figure 3.5: 11-bus distribution system diagram in ATP-EMTP.
CHAPTER 3. BASIC METHODOLOGY

It can be readily seen from the plots that the non-zeros do appear in the terminal nodes (node 4, 5, 14, 15 correspond to the A&B phases of Bus 2&6) and proportional relationship between the pairs of equivalent injections are valid. In addition, the decision making diagram can be easily utilized to identify that the fault is a double phase to ground fault.

3.1.4 Building the Linear Estimation Equations

As shown in (3.1) and the supporting simulation verification, it is possible to estimate the equivalent injections by a simple matrix-vector product of $Y_{bus}$ and $\Delta V$, which however requires full observ-
ability of the system, i.e. the pre and post fault voltages at all buses are assumed to be available and synchronized. Unfortunately, this is not a very realistic assumption for most power systems today. Thus, this assumption is relaxed by considering that the system is partially observable with voltage phasors known at only $m$ out of $N$ buses. Voltage phasors at these $m$ buses can be measured by Phasor Measurement Units (PMUs) placed at these buses or derived from voltages measured by PMUs at neighboring buses and current flow measurements on lines connecting these neighbors. This will lead to the following underdetermined version of nodal equations for the system:

$$Z_m I = \Delta V_m$$

(3.9)

where $Z_m$ and $\Delta V_m$ are extracted from $Z_{bus}$ and $\Delta V$ such that their rows correspond to the $m$ observable buses. In order for this complex-valued equation to be solved by existing sparse recovery algorithms, it is converted into a larger set of real equations as described in [30] and shown below.

$$X = \begin{bmatrix}
\text{real}(Z_m) & -\text{imag}(Z_m) \\
\text{imag}(Z_m) & \text{real}(Z_m)
\end{bmatrix}$$

(3.10)

$$y = \begin{bmatrix}
\text{real}(\Delta V_m) \\
\text{imag}(\Delta V_m)
\end{bmatrix}$$

(3.11)

As a result, the following real-valued estimation problem is obtained:

$$y = X \beta$$

(3.12)

The solution vector $\beta$ will contain two sub-vectors, namely the real and imaginary parts of the equivalent injections.
CHAPTER 3. BASIC METHODOLOGY

3.2 Sparse Estimation Algorithm

The linear equation given by (3.12) is underdetermined and would normally have infinite solutions. However, it could be proved that when the number of non-zeros in $\beta$ is smaller than half of the number of rows in $X$, its sparsest solution will be unique \[31\]. The sparsest solution for fault in-between line terminals could only be the one with equivalent bus injections because otherwise any sparser injections would indicate a fault on a bus. Therefore the most straightforward formulation for solving (3.12) with a sparsity constraint is:

$$\min \| \beta \|_0 \text{ s.t. } y = X\beta$$

(3.13)

where the $L_0$ norm of the solution $\beta$ is defined as:

$$\| \beta \|_0 = \# \{ i = 1, 2, ...N | \beta_i \neq 0 \}$$

(3.14)

Since this $L_0$ minimization formulation is non-convex and NP-hard as it involves combinatorial optimization, the authors turn to the formulation of $L_1$ minimization because $L_1$ norm is proved to be a convex relaxation of $L_0$ norm. A popular $L_1$ norm minimization formulation is Lasso (least absolute shrinkage and selection operator) \[32\]:

$$\hat{\beta} \in \arg \min_{\beta \in \mathbb{R}^p} \frac{1}{2} \| y - X\beta \|_2^2 + \lambda \| \beta \|_1$$

(3.15)

where the tuning of parameter $\lambda \geq 0$ leads to solutions with different sparsity. One efficient algorithm for solving this formulation is LARS (Least Angle Regression and Shrinkage) which iteratively selects a predictor with an equal correlation criterion \[33\]. The merit of this algorithm is that it is parameter-free and requires little computation time. In the following sections the formulation (3.15) and its solution through LARS algorithm will be addressed as basic LARS-Lasso method so as to be distinguished with its alternations.
CHAPTER 3. BASIC METHODOLOGY

The basic procedure for LARS is as follows, it can be concluded that the complexity of the LARS algorithm is $O(N)$, that is, it features linear complexity.

1. Start with an all-zero initial coefficient vector, and find the predictor (column in $X$) most correlated with the response;

2. Take the largest step possible in the direction of this predictor until some other predictor, has as much correlation with the current residual;

3. Proceed in a direction equiangular between the two predictors until a third variable earns its way into the “most correlated” set;

4. Then proceed equiangularly between the former three predictors, that is, along the “least angle direction”, until a fourth variable enters, and so on.

3.3 Measurement Placement for Unique Solution

The dictionary matrix in (3.15) should satisfy certain conditions in order for (3.15) to be solvable, and this condition can be used in placing the voltage measurements. Based on the KKT optimality conditions for (3.15), a sufficient condition for its unique solution is introduced in [34]. Firstly the concept of equicorrelation set $\epsilon$ is developed as follows:

$$\epsilon = \{i \in \{1, \ldots, p\} : |X_i^T(y - X_\hat{\beta})| = \lambda\} \quad (3.16)$$

In other words, $\hat{\beta}_\epsilon$ is the set containing only the non-zero entries of the solution $\hat{\beta}$. Then the condition for unique solution will be given by the following theorem:

**Lemma 1** For any $y$, $X$ and $\lambda > 0$, if $\text{null}(X_\epsilon) = 0$, or equivalently $\text{rank}(X_\epsilon) = |\epsilon|$, then the Lasso solution is unique.

In order to apply this condition for the placement of PMUs, the columns of the dictionary matrix $X$ and the variables in $\beta$ are divided into groups $g_1, g_2, \ldots, g_L$ where $L$ is the total number of lines and group $g_i$ contains the indices of columns in $X$ (also the indices of entries in $\beta$) corresponding to
CHAPTER 3. BASIC METHODOLOGY

the terminal buses of the $i_{th}$ line. Assuming that the fault occurs at one line at a time, the non-zero entries in the solution vector will appear in only one of the groups. To guarantee that the Lasso problem for each line has a unique solution, $\forall i \in 1, 2, ..., L$ the submatrix $X_{gi}$ should have full column rank. Note that if the positive sequence network of a system is to be studied, the submatrix extracted from $Z_m$ according to the grouping of $g_1, g_2, ..., g_L$ will only have two column vectors (or three for teed lines). If the full three phase model is used, the submatrix will contain six column vectors corresponding to the terminal buses in phases A, B, and C. However, even for this case, only two single phase columns need to be compared due to the inherent independence between columns of different phases.

Intuitively speaking, for two normalized column vectors if the entries at the same row are different from each other then it will be sufficient to consider them to be independent of each other. However, the bus impedance matrix is usually ill-conditioned with highly correlated columns, thus the numerical threshold for two entries to be different has to be large enough for reliable performance of Lasso solvers. This is accomplished by assigning $th = 50\%$ of the maximum difference between two columns as the threshold. As an example, for the $j_{th}$ line from bus $k$ to bus $m$ the following operation is performed:

$$d(:, j) = |Z_{bus}(; k) - Z_{bus}(; m)|$$

$$s(:, j) = \{i \in \{1, ..., N\} : d(i, j) > th \cdot \max(d(:, j))\}.$$  

This relationship can be summarized as a binary matrix $A_1$ in which each row is related with each branch and each column with each bus in the system.

$$A_1(i, j) = \begin{cases} 
1 & \text{if } i \in s(:, j) \\
0 & \text{otherwise.}
\end{cases}$$

17
CHAPTER 3. BASIC METHODOLOGY

Since a given bus voltage can be measured either directly by a PMU installed at the bus or indirectly by a PMU at one of its neighboring buses, similar to the procedure for optimally placing PMUs for state estimation [35], the $N \times N$ ($N$ is the number of buses) bus incidence matrix $A_2$ is defined as:

$$A_2(i,j) = \begin{cases} 1 & \text{if bus } i \text{ is connected with bus } j \\ 0 & \text{otherwise.} \end{cases}$$

(3.20)

In order to obtain a unique solution using the minimum number of PMUs the following binary integer optimization problem will be formulated, where $x_i = 1 (i = 1, 2, \ldots, N)$ implies an installed PMU at bus $i$, and the non-zero entries in the product $A_2x$ represent the buses whose voltages can be obtained based on the chosen PMU placement.

$$\min \sum_{i=1}^{N} x_i$$

(3.21)

$$\text{s.t. } A_1 A_2 x \geq \hat{1}, x \text{ is binary.}$$

(3.22)

The solution of this optimization will yield only partially observable system, unlike the case of PMU-based state estimation which requires full network observability.
Chapter 4

Alternations Improving the Method

Upon the description of the proposed method in 3, several alternations have been made aiming to improve the performance of the method in terms of reliability, robustness and efficiency.

4.1 Overlapping Group Lasso

In addition to the basic formulation of Lasso optimization, an alternative formulation is also considered, where the structural feature of the solution in addition to sparsity is taken into account, that is, the non-zero injections in the solution only appear at the terminal buses of a line. Therefore the solution is sparse not only element-wise but also group-wise. Recalling 3.3 where the buses in a system with $L$ lines can be grouped by $g_1, g_2, \ldots, g_L$, it is recognized that these groups are overlapping since a bus in transmission system is usually connected to more than one line. Therefore, incorporating the penalty on group sparsity leads to the following formulation [36]:

$$\hat{\beta} \in \arg\min_{\beta \in \mathbb{R}^p} \frac{1}{2} \| y - X\beta \|^2_2 + \lambda_1 \|\beta\|_1 + \lambda_2 \sum_{i=1}^{L} \| \beta_{g_i} \|_2$$  \hspace{1cm} (4.1)

This reformulated problem can be solved by using the “CVX”, a software package for specifying and solving convex programs [37,38].
CHAPTER 4. ALTERNATIONS IMPROVING THE METHOD

4.2  Second Stage OLS Estimation

4.2.1  Criteria for Valid Solution

After Lasso or overlapping group Lasso solvers return the solution vector for (3.15) or (4.1), the following criteria are used to check the validity of the solution.

1. After thresholding the absolute value of the entries in the solution with a small number $\epsilon$, the sparsity of the solution vector belongs to either 2, 4, 6 or 3, 6, 9 where the latter case indicates fault on a teed circuit line.

2. The indices of “non-zero” entries in the solution belongs to one of the groups of indices $g_1$, $g_2$, …, $g_L$.

3. The ratio between the non-zeros on each bus should satisfy either (3.4) or (3.8).

4.2.2  OLS Backup Estimation Routine

If the solution does not satisfy all the above conditions, then it will be necessary to fix the result based on whatever Lasso or overlapping group Lasso yields. Fortunately when these two solvers fail to give the correct coefficients in the solution, the support of the invalid solutions do contain the indices corresponding to at least one of the faulted line terminals. As a result, several groups of indices from the set $g_1$, $g_2$, …, $g_L$ overlapping with the support of the invalid solution are to be considered as suspect indices. Then the valid solution can be computed by a second stage unregularized ordinary least square estimation which has also been suggested in [33] under the name “LARS-OLS hybrid”. In the case of the proposed method, this is done by extracting the columns of the dictionary matrix $X$ using the suspected group indices (denoted by $s$) and solving an overdetermined linear equation via (4.2).

$$\hat{\beta}_s = (X_s^T X_s)^{-1} X_s^T y$$  (4.2)
CHAPTER 4. ALTERNATIONS IMPROVING THE METHOD

The resulting $\hat{\beta}_s$ with the least error will be the final solution. This second stage backup evaluation can be regarded as a reliable protection for the proposed method which might not work well under certain noises.

4.3 Extended Robust Formulation and Correction Routine

4.3.1 Extended Robust Formulation

Since chances are individual measurement unit in the system may fail or suffer from corruption, it is considered necessary to investigate the robustness of the proposed method against such gross error. This leads us to an extended Lasso formulation in [39], which incorporates sparse missing or grossly corrupted measurements into (3.15). Below is the extended version of the underdetermined linear equation (3.12), where $\hat{e}$ stands for sparse gross errors added to the correct voltage measurements.

$$
y = X\beta + e = \begin{bmatrix} X & I \end{bmatrix} \begin{bmatrix} \beta \\ e \end{bmatrix}
$$

(4.3)

Using (4.3), the only difference in the solution routine with the basic LARS-Lasso is the replacement of the original dictionary matrix $X$ with the extended matrix $\begin{bmatrix} X & I \end{bmatrix}$. In this way, if individual measurement units fail or happen to be inaccurate, the $e$ vector will have corresponding non-zero entries, the locations and values of gross errors along with correct equivalent injections can be identified simultaneously.

4.3.2 Recursive Solution Routine with Correction of Erroneous Measurement(s)

To achieve better performance using the extended robust formulation, the estimated gross error vector can be recycled to correct the measurements backwards, a corresponding recursive routine is designed as in Fig. 4.1. The initially corrupted measurement vector will be updated with the estimated $e$ and then used for the next round of Lars routine until no gross error is detected. This approach helps improving the final accuracy of the fault location even though it comes with the
increase of computation time which is not so critical since fault location is an offline application and single round of Lars takes little time to run.

![Flowchart for the recursive estimation](image)

**4.4 Parallel Implementation**

**4.4.1 Problem Decomposition**

In terms of efficiency of the proposed method, although it has been found that LARS has linear complexity, as the scale of power system increases, it’s worried that the computation speed of fault location program will drop below the satisfactory limit for the fault to be repaired in time. Thus investigations are made on how the sparse estimation problem can be decomposed allowing for solution using multiple processors running in parallel. Based on [40], there can be three types of data distribution for the dictionary matrix namely row block distribution, column block distribution and general block distribution are illustrated, as shown in Fig. 4.2 below.

Among the three distribution scenarios, it’s considered that column block distribution is the most appropriate one for parallelizing the original Lasso problem because in the application of power system fault location each column in the dictionary matrix represents a bus in the network and
aggregations of buses form sub-networks which is a common way of decomposing large power networks.

To decompose the original lasso problem into smaller problems, it is necessary to take into account again the notion of groups of indices. For each line in the network, there are four corresponding column indices in the dictionary matrix and this four indices form a group. For a network with \( L \) lines, there will be \( L \) groups of indices. Thus as shown in the column block distribution scenario, each parallel processor can take charge of a subset of the dictionary matrix’s columns which is the union of several groups of indices corresponding to several lines. As with the overlapping Lasso formulation, the subsets of columns are overlapping since the terminals of a line cannot be assigned to different groups.

When the problem is decomposed using the column block distribution, it can be easily seen that the sub-problems are independent with each other. In other words, the sub-problems are embarrassingly parallel \([41]\). Take Fig. 4.3 as an example, there are three pairs of estimation problems to solved in parallel: \((A_1, b)\), \((A_2, b)\) and \((A_3, b)\). Each one of \(\{A_1, A_2, A_3\}\) covers roughly \(L/n_{\text{Worker}}\), \((n_{\text{Worker}} = 3)\) groups of lines, the tie-line between two areas will correspond to the overlapping columns between two sub-matrices.

Assume the sparse injections exist in the \(x_1\) part, then the first sub-problem can be solved smoothly by the first processor, resulting in the same sparse solution with the original problem, but in the meantime the other sub-problems will run-into exhaustive search or exit halfway due to internal detection of discrepancies because there is no sparse solution for them. To achieve the goal
CHAPTER 4. ALTERNATIONS IMPROVING THE METHOD

Figure 4.3: Example of Column Block Distribution

of reducing computation time, there should be a communication mechanism between processors so
that once a single processor is done with its estimation other processors will be told to stop.

4.4.2 Parallelism Attempted

Based on the sequential Matlab code of the Lars algorithm [42] for solving Lasso, the author has
developed 4 parallel implementations of the algorithm using Matlab Parallel Computation Toolbox
(PCT) [43], Message Passing Interface (MPI) [44], Open Multi-Processing (openMP) [45] and
GPU [46] respectively. The MPI and openMP implementations are modified from the sequential C
code of LARS which uses the GNU Scientific Library (GSL) [47] for defining and calculating vectors
and matrices. The main reason for using the GSL library is that it’s built within the gnu-4.8.1 compiler
and it provides an interface to the Basic Linear Algebra Subprograms (BLAS) [48] operations which
apply to vector and matrix objects. For set operations such as union and set difference the standard
library (STL) [49] has been applied.

The following Fig. 4.4 identifies the development path and supporting libraries for each version.
Among the four parallel versions, Matlab PCT, MPI and openMP versions are based on the column
block decomposition of the original problem, therefore they are all embarrassingly parallel, each
thread/worker uses part of the dictionary matrix to apply the Lars algorithm and one of them will
eventually get the correct solution. The GPU version is written in Compute Unified Device Architec-
ture (CUDA) C [50], and GPU-accelerated libraries namely cuBLAS [51] and Thrust [52] which are
analogous to the CPU based GSLBLAS and STL libraries are utilized.
Among the 4 implementations, the Matlab PCT is only used for prototyping the problem decomposition and the SPMD (single program multiple data) scheme which is also used in the MPI implementation. Ans for the MPI version, it’s later found out that the communication desired is incompatible with MPI’s broadcast function. The MPI_Bcast() can only come into effect when it’s called before the use of the broadcasted variable and the root of the broadcast has to be assigned without knowing which thread can generate the correct sparse solution. Therefore in the MPI version the overall timing is also not determined by the “successful” thread but the thread who takes the longest time, nothing can be done to tell other threads to stop when one thread has succeeded. Therefore, neither Matlab PCT or MPI implementation will used for the efficiency test.

On the other hand, due to the inherent difference between the architecture of multi-core CPU and GPU, it’s not promising to transplant the solution routine on each core of the MPI or openMP implementation to each thread in GPU. In this light, the reasonable way is to take advantage of massive GPU threads to accelerate the matrix/vector operations while keeping the framework of
CHAPTER 4. ALTERNATIONS IMPROVING THE METHOD

the sequential C version unchanged. The author has resorted to the cuBLAS library which is a GPU-accelerated version of the complete standard BLAS library. Similarly the use of STL library has been replaced by using the its GPU analogous library named Thrust. Both cuBLAS and Thrust functions implicitly create objects on GPU and do the computation without user-defined kernel functions. Unfortunately although cuBLAS functions can be called by use-defined kernel functions, Thrust does not allow so, hence even though it’s fairly convenient to use these two libraries, there is no freedom in testing different grid and block sizes.

As for the openMP version, the most important merit of it is its use of shared memory instead of protocol based communication, even though this merit comes with the limit of computation resource: sheer openMP program can only use the cores equipped for one compute node whereas there is no such limit for MPI. The broadcast from the “successful” thread to others is easily implemented by changing the value of a shared variable so that the loop in other threads will break. OpenMP can control the overall run time by the “successful” thread which is the fastest thread among all.
Chapter 5

Simulation Results

To generate an overall picture of the method’s applicability and performance (accuracy, efficiency, robustness), various tests have been performed from different perspectives. The accuracy of the estimation is evaluated by the error in fault distance. Assign $D$ as the total length of the faulted line, $d$ as the distance from the fault point to the terminal bus where the equivalent injection is larger, and $r_{esti}$ and $r_{true}$ as the estimated and “true” ratio between the equivalent injections, the relative fault distance error can be derived by (5.3):

\[
\begin{align*}
    d_{esti} &= \frac{D}{r_{esti} + 1} \quad (5.1) \\
    d_{true} &= \frac{D}{r_{true} + 1} \quad (5.2) \\
    err &= \frac{\|d_{esti} - d_{true}\|}{\|d_{true}\|} = \frac{\|r_{true} - r_{esti}\|}{r_{esti} + 1} \quad (5.3)
\end{align*}
\]

In the following sections, firstly the universal applicability of the method to different fault types and line types are verified through an unbalanced test case as well as an “teed” line in the IEEE 118 bus test case from [54]. Then the performance of the proposed method and its variations are evaluated using extensive simulation in the 118 bus test case. A 300 bus test case as well as a 3395 bus test case will also be used for efficiency test of the parallel implementation.
CHAPTER 5. SIMULATION RESULTS

5.1 Faults in the Unbalanced Test Case

Using the same unbalanced ATP test case as in Section 3.1.3, the applicability of the proposed method to identifying faults of various types on both balanced and unbalanced lines has been validated. For the 10 line sections in the test system (Fig. 5.5), all five types of fault (single to three phase, grounded and ungrounded) are simulated and the equivalent injections are estimated using full measurements of all buses because the system is too small to apply the optimized PMU placement. The measurement noise and gross error is not taken into consideration in this case, since in later subsections these will be discussed for larger test cases. The relative error of the estimated distance for all simulated faults are summarized in Table 5.1 where “SLG, DL, DLG, 3L, 3LG” stand for single phase to ground fault, double phase short circuit fault, double phase to ground fault, three phase short circuit fault and three phase to ground fault respectively. The fault resistance is set to be 0.1 ohm.

<table>
<thead>
<tr>
<th>Section ID</th>
<th>SLG</th>
<th>DL</th>
<th>DLG</th>
<th>3L</th>
<th>3LG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sec 1</td>
<td>3.86e-5</td>
<td>6.75e-5</td>
<td>1.12e-4</td>
<td>7.57e-5</td>
<td>2.41e-5</td>
</tr>
<tr>
<td>Sec 2</td>
<td>2.10e-5</td>
<td>1.06e-5</td>
<td>1.18e-5</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sec 3</td>
<td>3.63e-5</td>
<td>2.30e-5</td>
<td>7.10e-6</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sec 4</td>
<td>5.40e-5</td>
<td>5.12e-5</td>
<td>5.06e-5</td>
<td>8.95e-5</td>
<td>1.38e-4</td>
</tr>
<tr>
<td>Sec 5</td>
<td>2.13e-5</td>
<td>3.30e-5</td>
<td>1.68e-5</td>
<td>7.35e-5</td>
<td>5.41e-5</td>
</tr>
<tr>
<td>Sec 6</td>
<td>1.37e-4</td>
<td>1.18e-4</td>
<td>7.44e-5</td>
<td>9.62e-5</td>
<td>9.24e-5</td>
</tr>
<tr>
<td>Sec 7</td>
<td>5.20e-3</td>
<td>2.00e-3</td>
<td>2.40e-3</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sec 8</td>
<td>6.16e-5</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sec 9</td>
<td>3.80e-3</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Sec 10</td>
<td>2.41e-5</td>
<td>5.84e-6</td>
<td>1.38e-5</td>
<td>3.49e-5</td>
<td>2.24e-5</td>
</tr>
</tbody>
</table>

From the statistics above it can be seen that the accuracy of the equivalent injection estimation is comparatively better in grounded faults than ungrounded fault. This is only a rough observation because the electromagnetic measurements provided by ATP have to be fitted into phasors, the performance of curve fitting also has effect on the final error. In any case, the proposed method can be regarded as accurate from practical point of view.
CHAPTER 5. SIMULATION RESULTS

5.2 Wide-area Cases

In the following sections the IEEE-118 bus test system for which only positive sequence model is available will be used firstly to verify the applicability of the proposed method to teed circuit fault, then examine the effectiveness of the proposed method in terms of accuracy, number of iterations and number of OLS backup evaluations under different levels of measurement noise. After implementing the binary optimization shown in (3.22), it came out that 24 PMUs need to be placed making 96 out of the 118 buses observable.

For faults on each of the 170 lines, changes in bus voltages are simulated by injecting a pair of proportional currents at the terminal buses commensurate with the description of equivalent injections in (3.4). The location results for all 170 lines in noise free condition as well as under different levels of Gaussian measurement noises are collected to generate statistical performance metrics. Both the LARS algorithm for Lasso and the CVX package are employed in Matlab environment using a 2.6GHz Intel i5 processor.

5.2.1 Teed Circuit Fault

To verify the equivalent three terminal injections described in Section 3.1.1.2. Bus 71 in the 118-Bus system is reduced as a teed point since it is a zero-injection bus with only bus 70, 72, and 73 as its neighbors. Faults are simulated on line section 70-71, 71-72 and 71-73 respectively and each fault can be located correctly. For instance, for a fault on line 70-71 at a point \((2/7)\)th of the line distance away from bus 70, the equivalent injections estimated by basic LARS-Lasso are:

\[
I_{70} = 383.0554 - j759.1415(\text{p.u.}) \quad (5.4)
\]
\[
I_{72} = 13.1256 - j24.9163(\text{p.u.}) \quad (5.5)
\]
\[
I_{73} = 47.1500 - j102.6877(\text{p.u.}) \quad (5.6)
\]
CHAPTER 5. SIMULATION RESULTS

Let \( z_{70-71} \), \( z_{71-72} \) and \( z_{71-73} \) represent the series line impedances and \( z_{70-f} \) represent the impedance from bus 70 to the fault point. Then, comparing the three injections leads to the following relationships:

\[
\frac{I_{72}}{I_{73}} = \frac{z_{71-72}}{z_{71-73}} = \frac{0.0446 + j0.1800}{0.0087 + j0.0454} \quad (5.7)
\]

\[
z_s = z_{70-71}z_{71-72} + z_{70-71}z_{71-73} + z_{71-72}z_{71-73} \quad (5.8)
\]

\[
I_f = I_{70} + I_{72} + I_{73} = 443.33 - j886.75 (p.u.) \quad (5.9)
\]

\[
z_{70-f} = \frac{I_{72}z_s}{I_fz_{71-73}} = 0.0025 + j0.0102 (p.u.) \quad (5.10)
\]

\[
\frac{z_{70-f}}{z_{70-71}} = 0.2863 \approx \frac{2}{7} \quad (5.11)
\]

Therefore, the location of the fault which is \((2/7)th\) of the total distance from bus 70 to the teed point (bus 71) is accurately estimated.

5.2.2 Comparison of Basic LARS-Lasso and Overlapping Group Lasso

The performance of the method is examined by comparing the results from basic LARS-Lasso formulation with overlapping group Lasso formulation in terms of 3 metrics: average computation time, average number of iterations, average error, and number of cases requiring second estimation among all fault cases. Since the software package CVX uses a more general purpose interior-point algorithm other than the LARS algorithm used for Lasso, it is much slower than LARS and the comparison of computation time is only for the sake of completeness, as it will be shown in later section how the efficiency will be using different number of computation cores. Graphic representation of comparison results are shown in Fig. 5.1 to Fig. 5.4.

It can be seen from Fig. 5.1 to Fig. 5.4 that the overlapping group Lasso works more stably looking from number of iteration, number of OLS backup evaluation under different levels of noise although it is not that highly accurate as basic Lars implementation, this may be due to the limited
CHAPTER 5. SIMULATION RESULTS

Figure 5.1: Computation Times Used by Basic LARS-Lasso and Overlapping Group Lasso.

Figure 5.2: Number of Iterations by Basic LARS-Lasso and Overlapping Group Lasso.

adjustment that can be made to the tolerance of CVX. However, if timing is to be placed importance on, overlapping group Lasso will be less favorable than basic LARS-Lasso.
CHAPTER 5. SIMULATION RESULTS

Figure 5.3: Error of Fault Distance by Basic LARS-Lasso and Overlapping Group Lasso.

Figure 5.4: Number of OLS Backup Estimations by Basic LARS-Lasso and Overlapping Group Lasso.
CHAPTER 5. SIMULATION RESULTS

5.2.3 Robustness of Extended LARS-Lasso Formulation Against Gross Error

To evaluate the robustness of the extended robust formulation, the same set of fault cases under Gaussian noises are solved using the extended formulation with one and two failed measurements (output equals zero) respectively and each case is run twice: one with and the other without the recursive routine. Hence there are four records of performance to be compared with that of the gross error free cases solved by basic LARS-Lasso. In each gross error-corrupted case the error vector can be accurately estimated and used for correcting the measurement in the recursive routine. Fig. 5.5 demonstrates the comparison of estimation error under different levels of noise. It can be concluded that the effect sparse gross errors have on the accuracy of the location method is within acceptable limits. Applying the recursive routine will help increase the accuracy towards that of the gross error-free cases.

![Comparison of Average Fault Distance Error](image)

Figure 5.5: Comparison of Fault Distance Error under One or Two Unit Failures
CHAPTER 5. SIMULATION RESULTS

5.3 Parallel Implementation Results

To evaluate the efficiency of the proposed method especially its parallel implementation, a 300 bus as well as 3395 bus test case are also used for simulations. The sequential and parallel programs are run on the discovery cluster offered by the Information Technology Services, Research Computing at Northeastern University. The queue used belongs to the node group “nodes10g2” whose technical specifications are as follows:

Table 5.2: Platform Information

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon CPU E5-2680 2.8GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td># logical cores</td>
<td>40</td>
</tr>
<tr>
<td>on each node</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>64GB</td>
</tr>
</tbody>
</table>

5.3.1 Sequential Run Time Benchmark

Before examining the speed-up gained from parallel implementation, the run time of Matlab and C++ sequential programs are record as below to serve as benchmark for efficiency test.

Table 5.3: Sequential Run Times

<table>
<thead>
<tr>
<th>nBus</th>
<th>Matlab Run Time</th>
<th>C++ Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>118</td>
<td>0.0245s</td>
<td>0.0116s</td>
</tr>
<tr>
<td>300</td>
<td>0.0981s</td>
<td>0.1153s</td>
</tr>
<tr>
<td>3395</td>
<td>89.4747s</td>
<td>213.5774s</td>
</tr>
</tbody>
</table>

5.3.2 Speed-up Using openMP

The openMP implementation is tested using a series number of cores ranging from 5 to 40 which is the maximum number of usable cores on a compute node. The speed-ups are shown in Fig 5.6 below.
CHAPTER 5. SIMULATION RESULTS

Figure 5.6: Comparison of Speed-ups for Different Cases using openMP

It can be seen that for large system as the 3395 bus test system, the accelerating effect openMP has on the efficiency is much more obvious than that for smaller cases. And as the number of cores goes up, the computation time for the 3395 bus case becomes more and more acceptable.

5.3.3 Speed-up Using GPU

Since the library functions automatically launch the kernel functions, it’s not possible to design the dimension of the grids, therefore there’s only one way of execution for the GPU version. The run times and corresponding speed-ups are shown in Table 5.4.

<table>
<thead>
<tr>
<th>nBus</th>
<th>GPU Run Time</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>118</td>
<td>0.4733s</td>
<td>0.02139</td>
</tr>
<tr>
<td>300</td>
<td>1.1700s</td>
<td>0.09852</td>
</tr>
<tr>
<td>3395</td>
<td>10.1567s</td>
<td>21.0283</td>
</tr>
</tbody>
</table>
CHAPTER 5. SIMULATION RESULTS

It seems odd that for smaller cases like the 118 and 300 bus case, the GPU version is much slower than the sequential C program. A close look at profiling results indicates that this is mostly caused by vector initialization and set operation functions using the Thrust library. The following is a extracted of the profiling result from which it could be seen that even the most time consuming cuBLAS function ("void gemv2T_kernel_val") is much faster than a Thrust function. Since there’s a lot set operation in the while loop and this kind of operation involves data dependency between elements of vectors, it’s not practical to write kernel functions to parallelize them. Therefore so far the most promising approach to parallelize the proposed method is through openMP.

### Table 5.5: Summary of Profiling Results

<table>
<thead>
<tr>
<th>Total Time %</th>
<th>Total Time</th>
<th>Total Calls</th>
<th>Avg. Time</th>
<th>Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>53.61%</td>
<td>384.56ms</td>
<td>8496</td>
<td>45.263us</td>
<td>void::thrust::system......</td>
</tr>
<tr>
<td>9.92%</td>
<td>71.172ms</td>
<td>33960</td>
<td>2.0950us</td>
<td>CUDA memcpyDtoH (device to host)</td>
</tr>
<tr>
<td>0.08%</td>
<td>450.72us</td>
<td>30</td>
<td>15.023us</td>
<td>void gemv2T_kernel_val</td>
</tr>
</tbody>
</table>

Despite of the non-ideal result, the profiling result does verify the efficiency of cuBLAS functions and the dynamic kernel launches by cuBLAS and Thrust function calls (the gridSize and blockSize varies each time). It’s fair to say that GPU can provide significant speed-up for floating point matrix operations while it’s not guaranteed to help with integer set operations.
Chapter 6

Conclusion

The major contribution of this work is the transformation of fault location problem of both two terminal lines and teed circuit lines into a sparse estimation problem or to be more specific, Lasso problem which can be solved effectively by the LARS algorithm. Using the condition for unique solution to the Lasso formulation, an optimal PMU placement scheme is designed which results in only partial observability of the system. Beside the basic LARS-Lasso implementation, the overlapping group Lasso formulation which customizes the penalty for sparsity using structural information of the solution is also built and implemented. In addition, an extended robust Lasso formulation designed to tackle sparse gross errors is applied and equipped with a recursive routine for correcting the discrepant measurements. To improve the method’s efficiency for large systems, parallel computation techniques are explored and implemented.

Using specifically cases, the proposed method’s universal applicability to different types of faults and lines are verified. The performance of basic LARS-Lasso implementation, overlapping group Lasso as well as Lasso with extended robust formulation are examined using wide-area test case. Furthermore, the result for parallel implementation base on openMP has revealed that for large systems, the proposed method can be accelerated so that the required time for locating the fault can be satisfactory.
CHAPTER 6. CONCLUSION

Future research topics might include the investigation of the method’s robustness against high noise levels using different amount of redundant measurements. The issue of parameter error in the dictionary matrix is also worthy of exploration.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


Appendix A

Appendix
APPENDIX A. APPENDIX

A.1

Matlab Code Solving Lasso Problem

This part of code implements the LARS with extended formulation. The codes for basic LARS and overlapping group Lasso are similar to this thus omitted.

```matlab
function stat = solver_robust_lars_fun(sig, errV_unit, err_level, lam_e, correct)
    global nBus nBranch BranchTypeInfo MeasuredBus Aeq Aeq_noQR Q Neighbour Zbus_red
    Zbus_f nG G isOpt LengthRatio I0 Result Flag Iter_Err

    nCase = 0;
    noise_ratio = -1*ones(nBranch,1);
    % Iter_Err = (-1)*ones(nBranch,2);
    Com_time = (-1)*ones(nBranch,1);
    Require_2nd_eval = false(nBranch,1);

    true_inj = [real(I0), imag(I0), LengthRatio*real(I0), LengthRatio*imag(I0)];
    true_inj = sort(true_inj); % increasing order

    [Q,R] = qr(Aeq_noQR);
    ext_Aeq = [R, lam_e*inv(Q)];
    %ext_Aeq = [Aeq, lam_e*eye(2*length(MeasuredBus))];

    stopCriterion = {};
    stopCriterion{1,1} = 'MSE';%'maxKernels';
    MSE = 1e-6;
    stopCriterion{1,2} = MSE;%1000
    for fault_sec_id = 1:1:nBranch;
        if BranchTypeInfo(fault_sec_id,3) > 0
            Flag(fault_sec_id) = -1;
            continue;
        end
    end
```
APPENDIX A. APPENDIX

nCase = nCase + 1;

% Flag(fault_sec_id) = 1;
Fault_sec = BranchTypeInfo(fault_sec_id,1:2);
I_s = sparse(Fault_sec.', [1;1], [LengthRatio*I0; I0], nBus, 1);
I = full(I_s);
deltaV = Zbus_f*I;
V_noise = sig.*randn(nBus,1); % normal distribution with a mean of 0 and a
                  % standard deviation of sig.
deltaV = deltaV+V_noise;
noise_ratio(nCase,1) = norm(V_noise)/norm(deltaV);
deltaV2 = deltaV(MeasuredBus);

beq_org = [real(deltaV2); imag(deltaV2)];
deltaV2(errV_unit,1) = deltaV2(errV_unit,1)*err_level;

beq = [real(deltaV2); imag(deltaV2)];
err_set = beq - beq_org;

beq = Q\beq;

Result{nCase,1} = fault_sec_id;
Result{nCase,2} = Fault_sec;
isValid = false;
unsolvableCase = 0;
existErr = true;
while existErr == true
t0 = clock;
sol = lars(beq, ext_Aeq, [], 'lasso',stopCriterion,[],0,1); % solve by
        lars
timel = etime(clock, t0);

[ttt,nIter] = size(sol);
Result{nCase,5} = nIter;
active_set = sol(1,nIter).active_set;
beta = sol(1,nIter).beta;

th = 2e-2*max(abs(beta));
inj_beta_idx = find(abs(beta)>th);
inj_nodes_idx = active_set(inj_beta_idx);
inj = beta(inj_beta_idx);

temp1 = find(inj_nodes_idx>2*nBus);
if ~isempty(temp1)
  err_val = inj(temp1)*lam_e;
  err_node_idx = inj_nodes_idx(temp1)-2*nBus;
  temp2 = find(inj_nodes_idx<=2*nBus);
  inj_nodes_idx = inj_nodes_idx(temp2);
  inj = inj(temp2);
  Result{nCase,10} = err_val;
  Result{nCase,11} = err_node_idx;
  error_vector = zeros(2*length(MeasuredBus),1);
  error_vector(err_node_idx) = err_val.';
  err_of_err = norm(error_vector-err_set)/norm(err_set);
  Result{nCase,12} = err_of_err;
end

existErr = false;

% beq = beq - Q\error_vector;

if correct == true
  beq = beq - Q\error_vector;
  existErr = true; % go back to do lars again
APPENDIX A. APPENDIX

end
else
    existErr = false;
end

end
Result(nCase,3) = inj_nodes_idx;
Result(nCase,4) = inj;

%% see if the inj is valid
if mod(length(inj_nodes_idx),2)==0
    isValid = checkInj(inj, inj_nodes_idx, nBus);
else
    isValid = false;
end

%% 2nd stage OLS
if ~isValid
    Flag(fault_sec_id) = -1;
    Require_2nd_eval(fault_sec_id) = true;
    [sel_idx, nOLS_test, inj_best, minResidual] = OLS_backup(inj_nodes_idx, nBus, beq, Aeq, Neighbour); % global Aeq Neighbour

    if sel_idx(1)~=Fault_sec(1) || sel_idx(2)~=Fault_sec(2) % if minResidual > norm(error_vector)
        disp(['case', num2str(nCase), ' for fault on ', num2str(Fault_sec(1)), '-', num2str(Fault_sec(2)), ' could not be solved through OLS!'])
        nIter = -1;
        unsolvableCase = -1;
        Flag(fault_sec_id) = -1;
    end
    Result(nCase,7) = sel_idx;
    Result(nCase,9) = nOLS_test;
    inj = inj_best.';
    Result(nCase,8) = inj;
else

end
Flag(fault_sec_id) = 1;

end

%% Record stats
Com_time(nCase) = timel;
Iter_Err(nCase,1) = nIter;
if unsolvableCase > -1
  unsolvableCase = genError3(true_inj, inj, LengthRatio);
  Result{nCase,6} = unsolvableCase;
end
Iter_Err(nCase,2) = unsolvableCase;

end

%% Process stats
total_2nd_eval = sum(Require_2nd_eval)
Iter_Err = Iter_Err(1:nCase,:);
Iter_Err_idx_1 = find(Iter_Err(:,1) > 0);
Iter_Err_idx_2 = find(Iter_Err(:,2) > 0);
valid_iter = Iter_Err(Iter_Err_idx_1,1);
valid_error = Iter_Err(Iter_Err_idx_2,2);
noise_ratio = noise_ratio(1:nCase,:);
ave_noise = mean(noise_ratio)
ave_iter = mean(valid_iter)
ave_err = mean(valid_error)
ave_time = mean(Com_time(1:nCase))
stat = [ave_noise, ave_iter, ave_err, ave_time, total_2nd_eval];
end
APPENDIX A. APPENDIX

A.2

Parallel C Code using openMP

This piece of attached code is the parallel C code using openMP for implementing LARS. The codes for sequential C and MPI implementation are similar to this thus omitted.

```c
#include <gsl/gsl_math.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_blas.h>
#include <gsl/gsl_linalg.h>
#include <gsl/gsl_errno.h>

// using namespace std;
```
#define MM_MAX_LINE_LENGTH 1025
#define MM_PREMATURE_EOF 12
#define PI 3.141592653589793238462643383279
#define MSE_th 0.001
#define RESOLUTION_OF_LARS 1e-12
#define REGULARIZATION_FACTOR 1e-11
#define MAX_ITER 50
#define BETA_TH 1e-3

double sum_gsl_vec(gsl_vector *vv)
{
    double result;
    result = 0;
    for (size_t k = 0; k < vv->size; k++)
    {
        result = result + gsl_vector_get (vv, k);
    }
    return result;
}

int new_mm_read_mtx_array_size(FILE *f, int *M, int *N)
{
    char line[MM_MAX_LINE_LENGTH];
    int num_items_read;
    /* set return null parameter values, in case we exit with errors */
    *M = *N = 0;

    /* now continue scanning until you reach the end-of-comments */
    do
    {
        if (fgets(line,MM_MAX_LINE_LENGTH,f) == NULL)
            return MM_PREMATURE_EOF;
    }while (line[0] == '%');
APPENDIX A. APPENDIX

```c
/* line[] is either blank or has M, N, nz */
if (sscanf(line, "%d %d", M, N) == 2)
    return 0;

else /* we have a blank line */
do
{
    num_items_read = fscanf(f, "%d %d", M, N);
    if (num_items_read == EOF) return MM_PREMATURE_EOF;
}
while (num_items_read != 2);

return 0;

int print_matrix(FILE *f, const gsl_matrix *m)
{
    int status, n = 0;

    for (size_t i = 0; i < m->size1; i++) {
        for (size_t j = 0; j < m->size2; j++) {
            if ((status = fprintf(f, "%.12f ", gsl_matrix_get(m, i, j))) < 0)
                return -1;
            n += status;
        }
        if ((status = fprintf(f, "\n")) < 0)
            return -1;
        n += status;
    }
    return n;
}
```

53
int print_vector(FILE *f, const gsl_vector *m)
{
    int status, n = 0;
    for (size_t j = 0; j < m->size; j++) {
        if ((status = fprintf(f, "%.12f\n", gsl_vector_get(m, j))) < 0)
            return -1;
        n += status;
    }
    return n;
}

int main(int argc, char **argv)
{
    int max_n_threads = omp_get_max_threads();
    printf("max #threads = %d\n", max_n_threads);

    int n_avail_procs = omp_get_num_procs();
    printf("# of available processors = %d\n", n_avail_procs);

    if (argc <2){printf("need exactly 1 argument for #threads\n");return 0;}

    int caseID = atoi(argv[1]);

    int startFaultCaseID = atoi(argv[2]);

    int endFaultCaseID = atoi(argv[3]);
    int nTotalFaultCases = endFaultCaseID-startFaultCaseID+1;

    int n_assigned_threads = atoi(argv[4]);
    omp_set_num_threads(n_assigned_threads);
    printf("#threads assigned = %d\n", n_assigned_threads);
APPENDIX A. APPENDIX

double start_time, end_time, exe_time;

//bool toldToStop = 0;
int othersStop = 0;

int tid;
gsl_set_error_handler_off();
/*=========================Build the sub-cases======================== */
//BUILD CASE===Lars_prob Lars_sub(caseID);

size_t nRow;
size_t nBr;

gsl_vector *b;
gsl_matrix *groupIdx;

FILE *f;
int m, n;
int row, col;
double entry;
/* Read Original Am */
gsl_matrix *Am;
char s[20];
sprintf(s, "Data/ieee%d/Aeq_%d.dat", caseID, caseID);
printf("reading \%s
", s);

f = fopen(s, "r");
if(f == NULL) {
    printf("ERROR: \%s does not exist, exiting.\n", s);
    exit(EXIT_FAILURE);
}
//mm_read_mtx_array_size(f, &m, &n);
new_mm_read_mtx_array_size(f, &m, &n);
APPENDIX A. APPENDIX

```
Am = gsl_matrix_calloc(m, n);
for(int i = 0; i < m*n; i++) {
    row = i % m;
    col = floor((double)i/m);
    fscanf(f, "%lf", &entry);
    gsl_matrix_set(Am, row, col, entry);
}
fclose(f);

/* Read groupIdx */
sprintf(s, "Data/ieee%d/groupIdx_%d.dat", caseID, caseID);
printf("reading %s\n", s);

f = fopen(s, "r");
if(f == NULL) {
    printf("ERROR: %s does not exist, exiting.\n", s);
    exit(EXIT_FAILURE);
}
//mm_read_mtx_array_size(f, &m, &n);
new_mm_read_mtx_array_size(f, &m, &n);
 groupIdx = gsl_matrix_calloc(m, n);
for(int i = 0; i < m*n; i++) {
    row = i % m;
    col = floor((double)i/m);
    fscanf(f, "%lf", &entry);
    gsl_matrix_set(groupIdx, row, col, entry);
}
fclose(f);

nRow = Am->size1;
//nTotalCol = Am->size2;
nBr = groupIdx->size1;
size_t nGroupPerWorker = floor((double)nBr/n_assigned_threads);
//beta = gsl_vector_calloc(nCol);
```
APPENDIX A. APPENDIX

```c
/*************************************************************************
* Decompose A =========
**************************************************************************/

#include omp barrier

//start_time = omp_get_wtime();
gsl_vector *time_array;
time_array = gsl_vector_calloc(nTotalFaultCases);

int iFaultCase;
for(iFaultCase = startFaultCaseID; iFaultCase <= endFaultCaseID; iFaultCase++)
{
    /* Read b */
    sprintf(s, "Data/ieee%d/beq_%d.dat", caseID, iFaultCase);
    printf("reading \%s\n", s);

    f = fopen(s, "r");
    if(f == NULL) {
        printf("ERROR: \%s does not exist, exiting.\n", s);
        exit(EXIT_FAILURE);
    }

    //mm_read_mtx_array_size(f, &m, &n);
    new_mm_read_mtx_array_size(f, &m, &n);
    b = gsl_vector_calloc(m);
    for (int i = 0; i < m; i++) {
        fscanf(f, "%lf", &entry);
        gsl_vector_set(b, i, entry);
    }
    fclose(f);
    #pragma omp parallel private(tid)
    {
        size_t nCol;
        gsl_matrix *A;
        bool no_xtx = 0;
        gsl_vector *beta;
    }
```
std::vector<size_t> active;
int nIter = 0;
int canPrintSol = 0;
tid = omp_get_thread_num();

nCol = 0;

std::vector<size_t> temp_idx;
size_t groupIdx_temp;
for(size_t t = (tid)*nGroupPerWorker; t < std::min(nBr,(tid+1)*nGroupPerWorker); t++)
{
    for(size_t p = 0; p < 4; p++)
    {
        groupIdx_temp = (size_t)gsl_matrix_get(groupIdx, t, p)-1;
        std::vector<size_t>::iterator it;
        it = std::find(temp_idx.begin(), temp_idx.end(), groupIdx_temp);
        if (it == temp_idx.end()) // not exist
        {
            temp_idx.push_back(groupIdx_temp);
            nCol += 1;
        }
    }
}
std::sort(temp_idx.begin(),temp_idx.end());
A = gsl_matrix_calloc(nRow,nCol);
beta = gsl_vector_calloc(nCol);

gsl_vector *A_temp_col;
A_temp_col = gsl_vector_calloc(nRow);

for(size_t p = 0; p < nCol; p++)
{
    gsl_matrix_get_col(A_temp_col, Am, temp_idx[p]);
APPENDIX A. APPENDIX

```c
    gsl_matrix_set_col(A, p, A_temp_col); // Aeq_temp = Aeq(:,Idx_worker{k,1});
  }
  gsl_vector_free(A_temp_col);
  temp_idx.clear();
  //gsl_matrix_free(Am);
  /*========================END OF CASE BUILDING*/
  //------------------------------*/

  #pragma omp barrier
  start_time = omp_get_wtime();

  /*========================SOLVE THE SUB-PROB*/
  //------------------------------*/
  std::vector <size_t> all_candidate;
  for(size_t i=0; i<nCol; i++) all_candidate.push_back(i);
  gsl_matrix *x;
  x = gsl_matrix_calloc(nRow, nCol);
  gsl_matrix_memcpy(x, A);
  gsl_vector *temp_col;
  temp_col = gsl_vector_calloc(nRow);
  double temp_sum_mean;
  double temp_access;
  for(size_t i = 0; i < nCol; i++) // i th column
  {
    gsl_matrix_get_col(temp_col, x, i); // x_ith_col->temp_col
    temp_sum_mean = sum_gsl_vec(temp_col);
    temp_sum_mean = temp_sum_mean/nRow;
    gsl_vector_add_constant(temp_col, -temp_sum_mean); // substract mean
    gsl_matrix_set_col(x, i, temp_col); // copy temp_col back to x
  }
  // x = x./ sx_rep;
  gsl_vector *sx;
```
APPENDIX A. APPENDIX

```c
sx = gsl_vector_calloc(nCol);
gsl_matrix *x2;
x2 = gsl_matrix_calloc(nRow, nCol);
gsl_vector *temp_col2;
temp_col2 = gsl_vector_calloc(nRow);
gsl_matrix_memcpy(x2, x);
gsl_matrix_mul_elements(x2, x);//x2<-x2.*x;
for(size_t i = 0; i < nCol; i++)// i th column
{
    gsl_matrix_get_col(temp_col, x, i);
    gsl_matrix_get_col(temp_col2, x2, i);
    temp_sum_mean = sum_gsl_vec(temp_col2);
    temp_sum_mean = sqrt(temp_sum_mean);
    //if (nCol*nCol > 1000000)
    if (nCol*nCol > 100000000)
    {
        if (temp_sum_mean < RESOLUTION_OF_LARS)
        {
            temp_sum_mean = GSL_P0SINF;
            //printf("sx(%lu)<RESOLUTION_OF_LARS, need to be erased from
            all_candidate\n",i);
            all_candidate.erase(all_candidate.begin()+i);//all_candidate
            = find(sx < GSL_P0SINF);
        }
    }
    gsl_vector_set(sx, i, temp_sum_mean);
    gsl_vector_scale(temp_col, 1/temp_sum_mean);//x = x/. sx_rep;
    gsl_matrix_set_col(x, i, temp_col);// copy temp_col back to x
}
gsl_matrix_free(x2);
gsl_vector_free(temp_col);
gsl_vector_free(temp_col2);
```
APPENDIX A. APPENDIX

```c
//

gsl_matrix *xtx;
//xtx = gsl_matrix_calloc(nCol, nCol);
//if (nCol*nCol > 10000000)
if (nCol*nCol > 1000000000)
{
    //printf("[%d]Too large matrix, lars will not pre-calculate xtx\n", tid);
    no_xtx = 1;
}
else
{
    xtx = gsl_matrix_calloc(nCol, nCol);
    //xtx = x^T*x;
    
    gsl_matrix *temp_c;
    temp_c = gsl_matrix_alloc(nCol,nCol);
    gsl_matrix_set_zero(temp_c);
    gsl_blas_dgemm(CblasTrans, CblasNoTrans, 1.0, x, x, 1.0, temp_c);
    gsl_matrix_memcpy(xtx, temp_c);
}

gsl_vector *y;
y = gsl_vector_calloc(nRow);

gsl_vector_memcpy(y, b);

double my;
my = sum_gsl_vec(b)/nRow;
gsl_vector_add_constant(y, -my);//y = b - my;
/*===============INITIALIZATION===============*/
active.clear();
//inactive = all_candidate;
```
APPENDIX A. APPENDIX

\[
\text{std::vector<size_t> inactive(all_candidate);} \\
gsl_vector *mu_a; \\
mu_a = gsl_vector_calloc(nRow); \\
gsl_vector_set_zero(mu_a); \\
\]

\[
gsl_vector *mu_a_plus; \\
mu_a_plus = gsl_vector_calloc(nRow); \\
gsl_vector *mu_a_OLS; \\
mu_a_OLS = gsl_vector_calloc(nRow); \\
\]

\[
//gsl_vector *beta; \\
//beta = gsl_vector_calloc(nCol); \\
gsl_vector_set_zero(beta); \\
gsl_vector *beta_new; \\
beta_new = gsl_vector_calloc(nCol); \\
gsl_vector *beta_OLS; \\
beta_OLS = gsl_vector_calloc(nCol); \\
\]

\[
gsl_vector *y2; \\
y2 = gsl_vector_calloc(nRow); \\
\]

\[
gsl_vector_memcpy(y2, y); \\
gsl_vector_mul(y2, y); \\
\text{double MSE = sum_gsl_vec(y2);} \\
\text{MSE = MSE/nRow;} \\
gsl_vector_free(y2); \\
\]

\[
gsl_vector *c; //correlation vector \\
c = gsl_vector_calloc(nCol); \\
\text{double C_max;} \\
//C_max = max(abs(c)); \\
\text{size_t C_max_ind;} \\
\text{std::vector<size_t> C_max_ind_pl;} \\
\text{std::vector<size_t> drop;} \\
\]
APPENDIX A. APPENDIX

```c
/*==========MAIN LOOP============*/
//othersStop = 0;
//MPI_Bcast(&othersStop, 1, MPI_INT, tid, MPI_COMM_WORLD);
while(nIter < MAX_ITER )//&& othersStop==0)
{
    if(othersStop == 1)
    {
        canPrintSol = 0;
        break;
    }

    if(MSE <= MSE_th)
    {
        printf("[%d]MSE <= MSE_th, break.\n", tid);
        canPrintSol = 1;
        break;
    }

    nIter += 1;
    //c = x^T*(y-mu_a);
    gsl_vector *y_mu_a;
    y_mu_a = gsl_vector_calloc(nRow);

    gsl_vector_memcpy(y_mu_a, y);
    gsl_vector_sub(y_mu_a, mu_a);

    gsl_vector_set_zero(c);
    gsl_blas_dgemv(CblasTrans, 1.0, x, y_mu_a, 1.0, c);
    gsl_vector_free(y_mu_a);

    double c_temp;
    gsl_vector *abs_c_inactive;
    abs_c_inactive = gsl_vector_calloc((size_t)inactive.size());
```
for(size_t k = 0; k < (size_t)inactive.size(); k++)
{
    c_temp = gsl_vector_get(c,inactive[k]);
    if(c_temp <0) c_temp = -c_temp;
    gsl_vector_set(abs_c_inactive, k, c_temp);
}

C_max = gsl_vector_max(abs_c_inactive);
C_max_ind = gsl_vector_max_index(abs_c_inactive);
C_max_ind = inactive[C_max_ind];//C_max_ind = inactive(C_max_ind);
//active = sort(union(active,C_max_ind));

std::vector<size_t>::iterator it;

it = std::find(active.begin(), active.end(), C_max_ind);
if(it == active.end())
{
    active.push_back(C_max_ind);
    //if(tid==10) fprintf(f_act_record, "[%d] %lu added to active

        set\n", nIter, C_max_ind);
}
std::sort(active.begin(), active.end());
//C_max_ind_pl = abs(c(inactive))>C_max-RESOLUTION_OF_LARS;
for(size_t k = 0; k < abs_c_inactive->size; k++)
{
    if(gsl_vector_get(abs_c_inactive,k) > (C_max-RESOLUTION_OF_LARS))
    {
        C_max_ind_pl.push_back(inactive[k]);//C_max_ind_pl =
        inactive(C_max_ind_pl);
        it = std::find(active.begin(), active.end(), inactive[k]);
        if (it == active.end())
        {
            active.push_back(inactive[k]);//active = sort(union(
                active, C_max_ind_pl));
            //if(tid==10) fprintf(f_act_record,"[%d]%lu added to
active set

\[ n_{\text{iter}}, \text{inactive}[k] \]\n
\}

\}

std::sort(active.begin(), active.end());
gsl_vector_free(abs_c_inactive);

std::vector<size_t> inactive_temp;

//inactive = setdiff(all_candidate, active);
std::set_difference(all_candidate.begin(), all_candidate.end(), active.begin(), active.end(), std::inserter(inactive_temp, inactive_temp.begin()));
inactive = inactive_temp;

std::vector<size_t>::iterator it2;

it2 = std::find(drop.begin(), drop.end(), C_max_ind);

if(!drop.empty() && it2 == drop.end())
{
    //active(find(active==C_max_ind))=[];
    //printf("[%d][%d]Dropped item and index of maximum correlation is not the same. But it is being ignored here...\n", tid, nIter);
    canPrintSol = 0;
    break;
}

if(!drop.empty())
{
    C_max_ind = 0;
    C_max_ind_pl.clear();
    //printf("size of drop is %lu, drop has %lu.%n",drop.size(), drop[0]);
}

//active = setdiff(active,drop);
APPENDIX A. APPENDIX

```cpp
std::vector<size_t> active_temp;
std::set_difference(active.begin(), active.end(), drop.begin(), drop.end(), std::inserter(active_temp, active_temp.begin()));
active = active_temp;

//inactive = sort(union(inactive,drop));
std::sort(drop.begin(), drop.end());
std::set_union(inactive.begin(), inactive.end(), drop.begin(), drop.end(), inactive_temp.begin());

//
gsl_vector *x_active_col;
x_active_col = gsl_vector_calloc(nRow);
gsl_matrix *xa;
xa = gsl_matrix_calloc(nRow, (size_t)active.size());
gsl_matrix *ga;
ga = gsl_matrix_calloc((size_t)active.size(), (size_t)active.size());

double s_temp_0;
gsl_vector *s;
if(active.empty())
{
    printf("[%d][%d] active is empty, maybe hard to allocate s\n", tid, nIter);
}
s = gsl_vector_calloc(active.size());

for(size_t k = 0; k < (size_t)active.size(); k++)
{
    double c_active_temp = gsl_vector_get(c, active[k]);
    if(c_active_temp >= RESOLUTION_OF_LARS)//s = sign(c(active));
    {gsl_vector_set(s, k, 1); s_temp_0 = 1}//s = 1;
    if(c_active_temp <= -RESOLUTION_OF_LARS)
```

66
\begin{verbatim}
if(c_active_temp < RESOLUTION_OF_LARS && c_active_temp > -
RESOLUTION_OF_LARS) {
    gsl_vector_set(s,k,0); s_temp_0 = 0;}//s = 0;
//printf("s = sign(c(active)), s[k] = %f\n",s_temp_0);
gsl_matrix_get_col(x_active_col, x, active[k]);//xa = x(:,active
).*repmat(s’T,n,1);
gsl_vector_scale(x_active_col, s_temp_0);
gsl_matrix_set_col(xa, k, x_active_col);
}
gsl_vector_free(x_active_col);
if(!no_xtx) {
    double ssT_temp;
    double xtx_temp;
    for(size_t i = 0; i < (size_t)active.size(); i++)//ga = xtx(
        active,active).*(s*s’T);
    {
        for(size_t j = 0; j < (size_t)active.size(); j++)
            {
                xtx_temp = gsl_matrix_get(xtx, active[i], active[j]);
                //printf("xtx(active[%d],active[%d]) = %lf\n",active[i],
                     active[j],xtx_temp);
                ssT_temp = gsl_vector_get(s,i)*gsl_vector_get(s,j);
                xtx_temp = xtx_temp*ssT_temp;
                //printf("after xtx_temp*s*s \n");
                gsl_matrix_set(ga, i, j, xtx_temp);
                //printf("ga(%d,%d) = %lf\n",i,j,xtx_temp);
            }
    }
}
gsl_vector_free(s);
\end{verbatim}
if(no_xtx)//ga = xa^T*xa;
{
    gsl_matrix_set_zero(ga);
    gsl_blas_dgemm(CblasTrans, CblasNoTrans, 1.0, xa, xa, 1.0, ga);
}
if(ga->size2<1)
{
    printf("[%d][%d] the nCol of ga is zero, hard to allocate Sga.\n", tid, nIter);
}
//invga = ga\eye(size(ga,1));
int ss;
gsl_matrix *invga = gsl_matrix_calloc(ga->size2, ga->size2);
gsl_permutation *perm = gsl_permutation_alloc(ga->size2);
//if(tid == 10 ) printf("[%d][%d] before inverting ga.\n", tid, nIter);

if(othersStop == 1)
{
    canPrintSol = 0;
    break;
}
gsl_matrix *ga_copy = gsl_matrix_calloc(ga->size1, ga->size2);
gsl_matrix_memcpy(ga_copy, ga);
gsl_matrix *Vga = gsl_matrix_calloc(ga->size2, ga->size2);
gsl_vector *Sga = gsl_vector_calloc(ga->size2);
int status = gsl_linalg_SV_decomp_jacobi(ga_copy, Vga, Sga);
if(status)
{ /* an error occurred */
printf("[%d][%d] Jacobi error occurred\n", tid, nIter);
canPrintSol = 0;
break;
}

//gsl_linalg_SV_decomp_jacobi(ga_copy, Vga, Sga);
if(gsl_vector_min(Sga)<1e-9)
{
    //printf("[%d][%d] ga is not invertible\n", tid, nIter);
canPrintSol = 0;
    break;
}
else
{
    double cond_ga;
    cond_ga = gsl_vector_max(Sga)/gsl_vector_min(Sga);
    if(cond_ga>1e9)
    {
        //printf("[%d][%d] cond(ga) is too large\n", tid, nIter);
canPrintSol = 0;
        break;
    }
}
gsl_matrix_free(ga_copy);
gsl_matrix_free(Vga);
gsl_vector_free(Sga);

gsl_linalg_LU_decomp(ga, perm, &ss);
gsl_linalg_LU_invert(ga, perm, invga);// THIS MIGHT CHANGE GA?
//if(tid == 10 ) printf("[%d][%d] after inverting ga.\n", tid, nIter
);
if(ga->size1<1)
{
    printf("[%d][%d] the nRow of ga is zero, hard to allocate
"
if (ga->size2 < 1) {
    printf("[%d][%d] the nCol of ga is zero, hard to allocate sum_invga_col&invga_row.\n", tid, nIter);
}
gsl_vector *invga_col;
invga_col = gsl_vector_calloc(ga->size1);
gsl_vector *sum_invga_col;
sum_invga_col = gsl_vector_calloc(ga->size2);
for (size_t k = 0; k < ga->size2; k++) {
    gsl_matrix_get_col(invga_col, invga, k);
    gsl_vector_set(sum_invga_col, k, sum_gsl_vec(invga_col));
}
double aa; // aa = \( \sum(\sum(\text{invga}))^{(-1/2)} \);
aa = sum_gsl_vec(sum_invga_col);
aa = sqrt(aa);
aa = 1/aa;
// printf("aa = %lf\n", aa);
gsl_vector_free(invga_col);
gsl_vector_free(sum_invga_col);
// wa = aa*sum(invga, 2);
gsl_vector *invga_row;
invga_row = gsl_vector_calloc(ga->size2);
if (invga->size1 < 1) {
    printf("[%d][%d] the nRow of invga is zero, hard to allocate wa.\n", tid, nIter);
}
gsl_vector *wa;
wa = gsl_vector_calloc(invga->size1);
double wa_temp;

for(size_t k = 0; k < ga->size1; k++)
{
    gsl_matrix_get_row(invga_row, invga, k);
    wa_temp = aa*sum_gsl_vec(invga_row);

    gsl_vector_set(wa, k, wa_temp);
}

for(size_t k = 0; k < ga->size1; k++)
{
    gsl_matrix_get_row(invga_row, invga, k);
    wa_temp = aa*sum_gsl_vec(invga_row);

    gsl_vector_set(wa, k, wa_temp);
}
gsl_vector_free(invga_row);
gsl_matrix_free(invga);
gsl_permutation_free(perm);

// ua = xa*wa;
gsl_vector *ua;
ua = gsl_vector_calloc(nRow);
gsl_vector_set_zero(ua);
gsl_blas_dgemv(CblasNoTrans, 1.0, xa, wa, 1.0, ua);
// test using Eq 2.7
if(xa->size2<1)
{
    printf("[%d][%d] the nCol of xa is zero, hard to allocate test_1
            .\n", tid, nIter);
}
gsl_vector *test_1; // test_1 = xa^T*ua;
test_1 = gsl_vector_calloc(xa->size2);
gsl_vector_set_zero(test_1);
gsl_blas_dgemv(CblasTrans, 1.0, xa, ua, 1.0, test_1);
if(test_1->size<1)
{
    printf("[%d][%d] the size of test_1 is zero, hard to allocate
            test_2.\n", tid, nIter);
}
gsl_vector *test_2; // test_2 = aa*ones(size(test_1));
test_2 = gsl_vector_calloc(test_1->size);
double test_1_2_temp;
for(size_t k = 0; k < xa->size2; k++)
{
    test_1_2_temp = gsl_vector_get(test_1, k);
    test_1_2_temp = test_1_2_temp - aa;
    if(test_1_2_temp<0) test_1_2_temp = -test_1_2_temp;
    gsl_vector_set(test_2, k, test_1_2_temp);
}

double test_1_2 = sum_gsl_vec(test_2);

double test_3;

gsl_blas_ddot(ua, ua, &test_3);

test_3 = sqrt(test_3);

test_3 = test_3 - 1;
    if(test_3 <0) test_3 = -test_3;

gsl_vector_free(test_1);

gsl_vector_free(test_2);
if(test_1_2 > RESOLUTION_OF_LARS*100 || test_3>RESOLUTION_OF_LARS*100)
{
    //printf("[%d][%d] Eq 2.7 test failure.\n", tid, nIter);
    canPrintSol = 0;
    break;
}

//a = x^T*ua;

gsl_vector *a;
a = gsl_vector_calloc(nCol);

gsl_vector_set_zero(a);
gsl_blas_dgemv(CblasTrans, 1.0, x, ua, 1.0, a);
```cpp
double c_inactive_temp;
double a_inactive_temp;
double tmp_1;
double tmp_2;
std::vector<double> tmp;

for(size_t k = 0; k < (size_t)inactive.size(); k++)
{
    c_inactive_temp = gsl_vector_get(c, inactive[k]);
    a_inactive_temp = gsl_vector_get(a, inactive[k]);
    tmp_1 = (C_max - c_inactive_temp) / (aa - a_inactive_temp);
    tmp_2 = (C_max + c_inactive_temp) / (aa + a_inactive_temp);

    //gsl_matrix_set(tmp_3, k, 1, tmp_1);
    //gsl_matrix_set(tmp_3, k, 2, tmp_2);
    if(tmp_1 > 0) tmp.push_back(tmp_1);
    if(tmp_2 > 0) tmp.push_back(tmp_2);
}

double gamm;

if(tmp.empty())
    gamm = C_max / aa;
else
    gamm = *std::min_element(tmp.begin(), tmp.end());

gsl_vector *d; //d = zeros(1,nCol);
d = gsl_vector_calloc(nCol);
gsl_vector_set_zero(d);
gsl_vector *tmpp; //tmpp = zeros(1,nCol);
tmpp = gsl_vector_calloc(nCol);
gsl_vector_set_zero(tmpp);
double d_temp;
double beta_temp;
std::vector<double> tmpp2;
```
double s_temp;

for(size_t k = 0; k < (size_t)active.size(); k++)
{
    double c_active_temp2 = gsl_vector_get(c, active[k]);
    if(c_active_temp2 >= RESOLUTION_OF_LARS) // s = sign(c(active));
        s_temp = 1;
    if(c_active_temp2 <= -RESOLUTION_OF_LARS)
        s_temp = -1;
    if(c_active_temp2 < RESOLUTION_OF_LARS && c_active_temp2 > -RESOLUTION_OF_LARS)
        s_temp = 0;
    //printf("active[k]=%d, s_temp = %f\n",active[k],s_temp);
    d_temp = s_temp*gsl_vector_get(wa, k);
    //printf("d[active[k] = s_temp.*wa = %lf\n”,d_temp);
    if(d_temp==0)//if (length(find(d(active)==0)))
    {
        //printf("[%d][%d] Something wrong with vector d: Eq 3.4\n", tid, nIter);
        canPrintSol = 0;
        break;
    }
    gsl_vector_set(d, active[k], d_temp);//d(active) = s.*wa;
}
for(size_t k = 0; k < (size_t)active.size(); k++)
{
    beta_temp = gsl_vector_get(beta, active[k]);
    d_temp = gsl_vector_get(d, active[k]);
    beta_temp = -beta_temp/d_temp;
    gsl_vector_set(tmpp, active[k], beta_temp); //tmp(active) = -1*
        beta(active)./d(active);
    if(beta_temp>0)
    {
        tmpp2.push_back(beta_temp);//tmpp2 = tmp(find(tmp>0));
APPENDIX A. APPENDIX

```cpp
//printf("%lf goes to tmpp2 in tmp2 = tmp(find(tmp>0));\n", beta_temp);

}  
}

drop.clear(); //drop = [];

double gamm_tilde = GSL_POSINF;

if(!tmpp2.empty())
{
    std::vector<double>::iterator min_tmpp2_p;
    min_tmpp2_p = std::min_element(tmpp2.begin(), tmpp2.end());

    double & min_tmpp2 = *min_tmpp2_p; // ????

    if(!tmpp2.empty() && gamm >= min_tmpp2)
    {
        gamm_tilde = min_tmpp2; // gamma_tilde = min(tmp2);
        for(size_t k = 0; k < nCol; k++) // drop = find(tmp == gamma_tilde);
        {
            if(gsl_vector_get(tmpp, k) == gamm_tilde)
            {
                drop.push_back(k);
                // if(tid==10) fprintf(f_act_record, "[%d] %lu added to drop at line: 808\n", nIter, k);
            }
        }
    }
}

tmpp2.clear();
gsl_vector_free(tmpp);

double min_gamm = gamm;
```
if(gamm > gamm_tilde) min_gamm = gamm_tilde;
gsl_vector_memcpy(mu_a_plus, mu_a);
gsl_blas_daxpy(min_gamm, ua, mu_a_plus);//mu_a_plus = mu_a + min( 
gamm,gamm_tilde)*ua;

gsl_vector_memcpy(beta_new, beta);
gsl_blas_daxpy(min_gamm, d, beta_new);//beta_new = beta + min(gamm, 
gamm_tilde)*d;

//active = setdiff(active,drop);
std::vector<size_t> active_temp2;
std::set_difference(active.begin(),active.end(),drop.begin(),drop.
end(),std::inserter(active_temp2, active_temp2.begin()));
active = active_temp2;
active_temp2.clear();

//inactive = setdiff(all_candidate,active);
std::vector<size_t> inactive_temp2;
std::set_difference(all_candidate.begin(),all_candidate.end(),active
.begin(),active.end(),std::inserter(inactive_temp2,
inactive_temp2.begin()));
inactive = inactive_temp2;
inactive_temp2.clear();

//printf("line 721:size of inactive is %lu.\n",inactive.size());
for(size_t k = 0; k < (size_t)drop.size(); k++)//beta_new(drop)=0;
{
    gsl_vector_set(beta_new, drop[k], 0);
}
double C_max_aa;
C_max_aa = C_max/aa;
gsl_vector_memcpy(mu_a_OLS, mu_a);
gsl_blas_daxpy(C_max_aa, ua, mu_a_OLS);//mu_a_OLS = mu_a + C_max/aa* 
ua;
APPENDIX A. APPENDIX

```c
827  gsl_vector_memcpy(beta_OLS, beta);
828  gsl_blas_daxpy(C_max_aa, d, beta_OLS);//beta_LS = beta + C_max/aa*d;
829
830  //MSE = sum((y-mu_a_OLS).ˆ2)/length(y);
831  gsl_vector *y_minus_mu_a;
832  y_minus_mu_a = gsl_vector_calloc(nRow);
833  gsl_vector_memcpy(y_minus_mu_a, y);
834  gsl_vector_sub(y_minus_mu_a, mu_a_OLS);
835  gsl_vector_mul(y_minus_mu_a, y_minus_mu_a);
836  double sum_y_mu_a = sum_gsl_vec(y_minus_mu_a);
837  MSE = sum_y_mu_a/nRow;
838
839  gsl_vector_free(y_minus_mu_a);
840  gsl_vector_free(d);
841  gsl_vector_free(ua);
842  gsl_vector_free(wa);
843  gsl_vector_free(a);
844  gsl_matrix_free(xa);
845  gsl_matrix_free(ga);
846
847  //update
848  gsl_vector_memcpy(mu_a, mu_a_plus);//mu_a = mu_a_plus;
849  gsl_vector_memcpy(beta, beta_new);//beta = beta_new;
850  double d_beta_temp;
851  gsl_vector *beta_backup;
852  beta_backup = gsl_vector_calloc(nCol);
853  gsl_vector_memcpy(beta_backup, beta);
854  if(MSE <= MSE_th)
855  {
856    //beta = beta(active)./sx(active);
857    gsl_vector_set_zero(beta);
858    for(size_t k = 0; k < (size_t)active.size(); k++)
859    {
860      d_beta_temp = gsl_vector_get(beta_backup, active[k]);
861      gsl_vector_set(beta, k, d_beta_temp);
862    }
```

d_beta_temp = d_beta_temp / gsl_vector_get(sx, active[k]);
gsl_vector_set(beta, active[k], d_beta_temp);
}

double temp_beta_element;
std::vector<size_t> active_backup(active);
for(size_t k = 0; k < (size_t)active_backup.size(); k++)
{
    temp_beta_element = gsl_vector_get(beta, active_backup[k]);
    if (temp_beta_element >= -BETA_TH && temp_beta_element <= BETA_TH)
    {
        std::vector<size_t>::iterator it_act;
        it_act = std::find (active.begin(), active.end(),
                             active_backup[k]);
        active.erase(it_act);
        gsl_vector_set(beta, active_backup[k], 0);
    }
}
active_backup.clear();
if(active.size() > 4)
{
    //printf("[%d] Invalid result despite of convergence.\n", tid);
    canPrintSol = 0;
    break;
}
canPrintSol = 1;
//othersStop = 1;//need to be broadcasted!
break;
}
gsl_vector_free(beta_backup);

//if(nIter>=MAX_ITER){printf("[%d] Reached nIter limit!\n", tid);}
```c
APPENDIX A. APPENDIX

gsl_vector_free(mu_a_plus);
gsl_vector_free(mu_a_OLS);
gsl_vector_free(beta_new);
gsl_vector_free(beta_OLS);
gsl_vector_free(c);
gsl_vector_free(mu_a);
gsl_vector_free(y);
gsl_matrix_free(x);
gsl_vector_free(sx);

if(!no_xtx) {gsl_matrix_free(xtx);}

if(!canPrintSol)
{
    gsl_matrix_free(A);
    //gsl_vector_free(b);
    gsl_vector_free(beta);
    //gsl_matrix_free(groupIdx);
    active.clear();
}

if(canPrintSol) // && !othersStop)
{
    othersStop = 1;

    end_time = omp_get_wtime();
    exe_time = end_time - start_time;
    printf("========Solved successfully by tid %d========\n",tid);

    printf("[\%d]\Thread Runtime = %f\n", tid, exe_time);
    printf("[\%d]Runtime = %f\n", tid, exe_time);
    //fprintf(f_record, "[\%d]Runtime = %f\n", tid, exe_time);
    //printf("[\%d]Runtime = %f\n", tid, exe_time);
```
printf("[%d]====Number of Iteration: %d====\n",tid, nIter);
for(size_t iii = 0; iii<active.size(); iii++)
{
    printf("[%d]------active_idx:%lu------beta[active_idx]:%6.8f

------\n",tid, active[iii], gsl_vector_get(beta,active[iii])
    );
}

gsl_matrix_free(A);
//gsl_vector_free(b);
gsl_vector_free(beta);
//gsl_matrix_free(groupIdx);
active.clear();
printf(""}

#pragma omp barrier
end_time = omp_get_wtime();
if(tid == 0)
{
    exe_time = end_time - start_time;
    printf("[%d] Runtime = %f\n", iFaultCase, exe_time);
    gsl_vector_set(time_array, iFaultCase-startFaultCaseID, exe_time);
}
}

gsl_vector_free(b);
}

//end_time = omp_get_wtime();
//exe_time = end_time - start_time;

printf("=============Average Runtime = %f\n", sum_gsl_vec(time_array)/
nTotalFaultCases);
APPENDIX A. APPENDIX

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <cuda_runtime.h>
// for std
#include <math.h>
#include <algorithm>
#include <string>
#include <vector>
// includes, cuda
#include <cublas.h>
#include <cupblas_v2.h>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/device_ptr.h>
```

A.3

CUDA C Code for GPU implementation

This piece of attached code is the CUDA C code used for GPU implementation.
APPENDIX A. APPENDIX

```c
#include <thrust/sort.h>
#include <thrust/merge.h>
#include <thrust/transform.h>
#include <thrust/reduce.h>
#include <thrust/scan.h>
#include <thrust/set_operations.h>
#include <thrust/iterator/discard_iterator.h>
#include <thrust/device_ptr.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/iterator/transform_iterator.h>
#include <thrust/iterator/permutation_iterator.h>
#include <thrust/extrema.h>
#include <thrust/functional.h>
#include <thrust/sequence.h>
#include <thrust/fill.h>

#define MM_MAX_LINE_LENGTH 1025
#define MM_PREMATURE_EOF 12
#define PI 3.141592653589793238462643383279
#define MSE_th 0.001
#define RESOLUTION_OF_LARS 1e-12
#define REGULARIZATION_FACTOR 1e-11
#define MAX_ITER 50
#define INF 1e12
#define BETA_TH 1e-3

// matrix indexing convention
#define id(m, n, ld) (((n) * (ld) + (m)))

int new_mm_read_mtx_array_size(FILE *f, int *M, int *N)
{
    char line[MM_MAX_LINE_LENGTH];
}
APPENDIX A. APPENDIX

```c
int num_items_read;
/* set return null parameter values, in case we exit with errors */
*M = *N = 0;

/* now continue scanning until you reach the end-of-comments */
do
{
    if (fgets(line,MM_MAX_LINE_LENGTH,f) == NULL)
        return MM_PREMATURE_EOF;
}while (line[0] == '%');

/* line[] is either blank or has M,N, nz */
if (sscanf(line, "%d %d", M, N) == 2)
    return 0;
else /* we have a blank line */
do
{
    num_items_read = fscanf(f, "%d %d", M, N);
    if (num_items_read == EOF) return MM_PREMATURE_EOF;
}
while (num_items_read != 2);
return 0;

struct MinusC: public thrust::unary_function<double, double>
{
    const double offset;
    __host__ __device__ MinusC(double _offset) :
        offset(_offset)
    {
    }
    __host__ __device__ double operator()(double x)
```
struct DivideByC: public thrust::unary_function<double, double>
{
    const double offset;
    __host__ __device__ DivideByC(double _offset) :
        offset(_offset)
    {
    }
    __host__ __device__ double operator()(double x)
    {
        return (double) x/offset;
    }
};

struct MultipleByC: public thrust::unary_function<double, double>
{
    const double offset;
    __host__ __device__ MultipleByC(double _offset) :
        offset(_offset)
    {
    }
    __host__ __device__ double operator()(double x)
    {
        return (double) x*offset;
    }
};

struct Inv: public thrust::unary_function<double, double>
{
    __host__ __device__ double operator()(double x)
    {
        return (double) 1.0 / x;
    }
}
APPENDIX A. APPENDIX

```cpp
struct Abs: public thrust::unary_function<double, double>
{
    __host__ __device__ double operator()(double x)
    {
        if(x<0) x = -x;
        return (double) x;
    }
};

struct SquareRoot: public thrust::unary_function<double, double>
{
    __host__ __device__ double operator()(double x)
    {
        return (double) sqrt(x);
    }
};

struct Square: public thrust::unary_function<double, double>
{
    __host__ __device__ double operator()(double x)
    {
        return (double) x*x;
    }
};

struct FilterIgnores: public thrust::unary_function<double, double>
{
    __host__ __device__ double operator()(double x)
    {
        if(x< RESOLUTION_OF_LARS)
            return INF;
        else
            return x;
    }
};

// main
```
APPENDIX A. APPENDIX

```c
int main(int argc, char** argv)
{

    if (argc <2){printf("need exactly 1 argument for #threads\n");return 0;}

    //double start_time, end_time, exe_time;

    int caseID = atoi(argv[1]);
    printf("Will solve nBus = %d case.\n", caseID);

    int startFaultCaseID = atoi(argv[2]);

    int endFaultCaseID = atoi(argv[3]);
    int nTotalFaultCases = endFaultCaseID-startFaultCaseID+1;
    //int nProcs = atoi(argv[2]);
    /*=========================BUILD THE PROBLEM======================== */

    bool no_xtx = 0;

    FILE *f;
    int m, n;

    double entry;
    /* Read A */
    char s[20];
    sprintf(s, "Data/ieee%d/Aeq_%d.dat", caseID, caseID);
    printf("reading %s\n", s);

    f = fopen(s, "r");
    if(f == NULL) {
        printf("ERROR: %s does not exist, exiting.\n", s);
        exit(EXIT_FAILURE);
    }
    //mm_read_mtx_array_size(f, &m, &n);

```
APPENDIX A. APPENDIX

```c
new_mm_read_mtx_array_size(f, &m, &n);

int nRow = m;
int nCol = n;

printf("nRow = %d, nCol = %d.\n", nRow, nCol);
//A = (double*)malloc(nRow*nCol * sizeof(double));
thrust::host_vector<double> A(nRow*nCol);

int row, col;

for(int i = 0; i < m*n; i++) {
    row = i % m;
    col = floor((double)i/m);
    fscanf(f, "%lf", &entry);
    //gsl_matrix_set(Am, row, col, entry);
    A[id(row, col, nRow)] = entry;
}

close(f);

thrust::host_vector<double> timearray(nTotalFaultCases);
int iFaultCase ;
for(iFaultCase = startFaultCaseID; iFaultCase <= endFaultCaseID; iFaultCase ++ )
{
    /* Read b */
    sprintf(s, "Data/ieee%d/beq_%d.dat", caseID, iFaultCase);
    printf("reading %s\n", s);
    f = fopen(s, "r");
    if(f == NULL) {
```

87
APPENDIX A. APPENDIX

221 printf("ERROR: $s does not exist, exiting.\n", s);
222 exit(EXIT_FAILURE);
223 }
224 //mm_read_mtx_array_size(f, &m, &n);
225 new_mm_read_mtx_array_size(f, &m, &n);
226 if(m!=nRow) {
227     printf("ERROR: length of b doesn’t match with nRow of A.\n", s);
228     exit(EXIT_FAILURE);
229 }
230 //b = gsl_vector_calloc(m);
231 //b = (double*)malloc(nRow * sizeof(double));
232 thrust::host_vector<double> b(nRow);
233
234 for (int i = 0; i < nRow; i++) {
235     fscanf(f, "%lf", &entry);
236     b[id(i,0,nRow)] = entry;
237 }
238 fclose(f);

240 /*====================INITIALIZATION OF TIMING AND CUBLAS

====================*/

242 //clock_t start=clock();
243 //double dtime;
244
245 // initialize cublas
246 cublasStatus_t status;
247 cublasHandle_t hd;
248
249 status = cublasCreate(&hd);
250
252 if (status != CUBLAS_STATUS_SUCCESS)
253 {
APPENDIX A. APPENDIX

fprintf(stderr, "!!!! CUBLAS initialization error\n");
return EXIT_FAILURE;
}
else printf("CUBLAS initializations success\n");

/*===========================SOLVE THE PROBLEM
   ==============================*/

// transfer host to device
thrust::device_vector<double> dx(A.begin(), A.end());
thrust::device_vector<double> dy(b.begin(), b.end());

clock_t start=clock();
double dtime;

thrust::device_vector<int> all_candidate(nCol);
// initialize all_candidate to 0,1,2,3, .... for(int i=0; i<nCol; i++)
    all_candidate.push_back(i);
thrust::sequence(all_candidate.begin(), all_candidate.end(),0);

// mx = mean(xin); x = xin - repmat(mx,size(xin,1),1);
thrust::device_vector<double> ones(nRow, 1);
thrust::device_vector<double> sumA(nCol, 0);

double* p_dx = thrust::raw_pointer_cast(&dx[0]);
double* p_dy = thrust::raw_pointer_cast(&dy[0]);
double* p_ones = thrust::raw_pointer_cast(&ones[0]);
double* p_sumA = thrust::raw_pointer_cast(&sumA[0]);

const double c1 = 1.0;
const double c0 = 0.0;
const double c_1 = -1.0;
cublasDgemv(hd, CUBLAS_OP_T, nRow, nCol, &c1, p_dx, nRow, p_ones, 1, &c0,
            p_sumA, 1);

thrust::transform(sumA.begin(), sumA.end(), sumA.begin(), DivideByC((
    double)nRow));

// nRow*nCol, make each row of B sumA[k]
thrust::device_vector<double> BsumA(nRow*nCol);
for(int k = 0; k < nCol; k++)// TOO SLOW!!!
{
    thrust::fill(BsumA.begin()+k*nRow, BsumA.begin()+(k+1)*nRow, sumA[k]);
}
double* p_BsumA = thrust::raw_pointer_cast(&BsumA[0]);
thrust::device_vector<double> dx_copy = dx;
double* p_dxcopy = thrust::raw_pointer_cast(&dx_copy[0]);
cublasDgemm(hd, CUBLAS_OP_N, CUBLAS_OP_N, nRow, nCol, &c1, p_dxcopy,
            nRow, &c_1, p_BsumA, nRow, p_dx, nRow);
dx_copy.clear();

// sx = sum(x.^2).^((1/2)); ignores = find(sx < RESOLUTION_OF_LARS); sx(ignores)=inf;
// x = x ./ repmat(sx,size(xin,1),1);
thrust::device_vector<double> sx = dx;
thrust::transform(sx.begin(), sx.end(), sx.begin(), Square());

double* p_sx = thrust::raw_pointer_cast(&sx[0]);
cublasDgemv(hd, CUBLAS_OP_T, nRow, nCol, &c1, p_sx, nRow, p_ones, 1, &c0,
            p_sumA, 1);

thrust::transform(sumA.begin(), sumA.end(), sumA.begin(), SquareRoot());
thrust::device_vector<double> ssx(sumA.begin(), sumA.end());

90
//thrust::device_vector<int> all_candidate_backup = all_candidate;
for (int k = 0; k < nCol; k++)
{
  if(sumA[k]<RESOLUTION_OF_LARS)
  {
    sumA[k] = INF;
    all_candidate.erase(all_candidate.begin()+k);
  }
  //thrust::transform(dx.begin()+k*nRow, dx.begin()+ (k+1)*nRow, dx. begin() + k*nRow, DivideByC(sumA[k]));
  //cublasDscal(hd, nCol, 1/sumA[k], p_dx, 1);
}
thrust::device_vector<double> forDivide(nCol*nCol,0);
for (int k = 0; k < nCol; k++) // TOO SLOW!!!
{
  forDivide[id(k,k,nCol)] = 1/sumA[k];
}
double* p_forDivide = thrust::raw_pointer_cast(&forDivide[0]);
thrust::device_vector<double> dx_copy2 = dx;
double* p_dxcopy2 = thrust::raw_pointer_cast(&dx_copy2[0]);
cublasDgemm(hd, CUBLAS_OP_N, CUBLAS_OP_N, nRow, nCol, nCol, &c1,
p_dxcopy2, nRow, p_forDivide, nCol, &c0, p_dx, nRow);
dx_copy2.clear();
sumA.clear();
one.clear();
sx.clear();

thrust::device_vector<double> xtx(1, 0);
//double* p_dxtx = thrust::raw_pointer_cast(&xtx[0]);
if (nCol*nCol > 100000000)
{
  printf("Too large matrix, lars will not pre-calculate xtx\n");
  no_xtx = 1;
}
```cpp
else {
    xtx.resize(nCol*nCol);
    //thrust::device_vector<double> xtx(nCol*nCol, 0);
    double* p_dxtx = thrust::raw_pointer_cast(&xtx[0]);
    //xtx = x^T*x;
    cublasDgemm(hd, CUBLAS_OP_T, CUBLAS_OP_N, nCol, nCol, nRow, &c1,
                p_dx, nRow, p_dx, nRow, &c0, p_dxtx, nCol);
}

//y = yin-mean(yin);
double mean_y;
    cublasDdot (hd, nRow, p_dy, 1, p_ones, 1, &mean_y);
    mean_y = mean_y/nRow;
    thrust::transform(dy.begin(), dy.end(), dy.begin(), MinusC(mean_y));
    ones.clear();

    /*==========INITIALIZATION==========*/
    thrust::device_vector<int> d_active;
    //active.clear();
    //inactive = all_candidate;
    thrust::device_vector<int> inactive=all_candidate;
    thrust::device_vector<double> mu_a(nRow, 0);
    thrust::device_vector<double> mu_a_plus(nRow, 0);
    thrust::device_vector<double> mu_a_OLS(nRow, 0);
    thrust::device_vector<double> beta(nCol, 0);
    thrust::device_vector<double> beta_new(nCol, 0);
    thrust::device_vector<double> beta_OLS(nCol, 0);
    //MSE = sum(y.^2)/length(y);
    thrust::device_vector<double> y2=dy;
    double* p_y2 = thrust::raw_pointer_cast(&y2[0]);
    thrust::transform(y2.begin(), y2.end(), y2.begin(), Square());
    double MSE;
```
cublasDasum(hd, nRow, p_y2, 1, &MSE);
MSE = MSE/nRow;
y2.clear();

thrust::device_vector<double> c(nCol);

//double C_max;
//C_max = max(abs(c));
//int C_max_ind;
thrust::device_vector<int> C_max_ind_pl;
thrust::device_vector<int> drop;
int nIter = 0;
bool canPrintSol = 0;

/*==============MAIN LOOP==============*/
while(nIter < MAX_ITER)
{
  if(MSE <= MSE_th)
  {
    printf("MSE <= MSE_th=%6.8f, break.\n", MSE_th);
    canPrintSol = 0;
    break;
  }
  const double cc1 = 1.0;
  const double cc0 = 0.0;
  nIter += 1;

  //c = x^T*(y-mu_a);
thrust::device_vector<double> y_mu_a(nRow);
thrust::transform(dy.begin(), dy.end(), mu_a.begin(), y_mu_a.begin(), thrust::minus<double>())
    , thrust::minus<double>())

//c = x^T*(y-mu_a);
double* p_y_mua = thrust::raw_pointer_cast(&y_mu_a[0]);
thrust::device_vector<double> c(nCol, 0);
double* p_c = thrust::raw_pointer_cast(&c[0]);

cublasDgemv(hd, CUBLAS_OP_T, nRow, nCol, &cc1, p_dx, nRow, p_y_mua,
    1, &cc0, p_c, 1);
y_mu_a.clear();

// [C_max,C_max_ind] = max(abs(c(inactive))); C_max_ind = inactive
    (C_max_ind);
thrust::device_vector<int> C_max_ind(1);
int* p_cmaxInd = thrust::raw_pointer_cast(&C_max_ind[0]);
thrust::device_vector<double> c_inact(inactive.size());

for(int k = 0; k < inactive.size(); k++)
{
    c_inact[k] = c[inactive[k]];
    if(c_inact[k] < 0) c_inact[k] = -c_inact[k];
}
const double* p_cinact = thrust::raw_pointer_cast(&c_inact[0]);

thrust::device_vector<double>::iterator iter_max = thrust:::
    max_element(c_inact.begin(), c_inact.end());

unsigned int position = iter_max - c_inact.begin();
double C_max = *iter_max;

C_max_ind[0] = inactive[position];

for(int k = 0; k < inactive.size(); k++)
{
    if(c_inact[k] >= C_max-RESOLUTION_OF_LARS && inactive[k]!=
        C_max_ind[0]) C_max_ind.push_back(inactive[k]);
}
//active = sort(union(active,C_max_ind));
// set_union returns an iterator C_end denoting the end of input

thrust::device_vector<int>::iterator C_end;

thrust::device_vector<int> new_active0(d_active.size()+C_max_ind.size());

C_end = thrust::set_union(d_active.begin(), d_active.end(),
    C_max_ind.begin(), C_max_ind.end(), new_active0.begin());

// shrink C to exactly fit output
new_active0.erase(C_end, new_active0.end());
d_active = new_active0;
new_active0.clear();

thrust::sort(d_active.begin(), d_active.end(), thrust::less<int>());

//inactive = setdiff(all_candidate, active);

thrust::device_vector<int>::iterator inact_end;
inact_end = thrust::set_difference(all_candidate.begin(),
    all_candidate.end(), d_active.begin(), d_active.end(), inactive.begin());
inactive.erase(inact_end, inactive.end());

//if ~isempty(drop) & length(find(drop==C_max_ind))==0

thrust::device_vector<int>::iterator iter;
iter = thrust::find(drop.begin(), drop.end(), C_max_ind[0]);

if(!drop.empty() && iter == drop.end())
{
    printf("[%d]Dropped item and index of maximum correlation is not
      the same. But it is being ignored here...\n", nIter);
    canPrintSol = 0;
    break;
}

if(!drop.empty())
C_max_ind.clear();

//active = setdiff(active, drop);
thrust::device_vector<int> new_active(d_active.size());
thrust::device_vector<int>::iterator act_end;
act_end = thrust::set_difference(d_active.begin(), d_active.end(), drop.begin(), drop.end(), new_active.begin());
new_active.erase(act_end, new_active.end());
d_active = new_active;
new_active.clear();

//inactive = sort(union(inactive, drop));
thrust::device_vector<int> new_inactive(inactive.size() + drop.size());
inact_end = thrust::set_union(inactive.begin(), inactive.end(), drop.begin(), drop.end(), new_inactive.begin());
new_inactive.erase(inact_end, new_inactive.end());
inactive = new_inactive;
new_inactive.clear();
thrust::sort(inactive.begin(), inactive.end(), thrust::less<int>());

if(d_active.empty())
{
    printf("[%d] active is empty, maybe hard to allocate s\n", nIter);
}

//s = sign(c(active)); % eq 2.10 xa = x(:, active).*repmat(s', n, 1);
% eq 2.4
thrust::device_vector<double> xa(nRow * d_active.size());
thrust::device_vector<double> ga(d_active.size() * d_active.size());
thrust::device_vector<double> s(d_active.size());
double s_temp_0;
//xa = x(:, active).*repmat(s’T, n, 1);
for(int k = 0; k < d_active.size(); k++)
APPENDIX A. APPENDIX

```cpp
{  
    double c_active_temp = c[d_active[k]];  
    if(c_active_temp >= RESOLUTION_OF_LARS) // s = sign(c(active));  
        {s[k] = 1; s_temp_0 = 1;}//s = 1;  

    if(c_active_temp <= -RESOLUTION_OF_LARS)  
        {s[k] = -1; s_temp_0 = -1;}//s = -1;  

    if(c_active_temp < RESOLUTION_OF_LARS && c_active_temp > -RESOLUTION_OF_LARS)  
        {s[k] = 0; s_temp_0 = 0;}//s = 0;  

    thrust::copy(dx.begin()+d_active[k]*nRow, dx.begin()+(d_active[k]+1)*nRow, xa.begin()+k*nRow);  

    thrust::transform(xa.begin()+k*nRow, xa.begin()+(k+1)*nRow, xa.begin()+k*nRow, MultipleByC(s_temp_0));
}

if(!no_xtx)//ga = xtx(active,active).*(s*s^T);  
{
    double ssT_temp;
    double xtx_temp;
    for(int i = 0; i < d_active.size(); i++)
        {
            for(int j = 0; j < d_active.size(); j++)
                {
                    xtx_temp = xtx[id(d_active[i], d_active[j], nCol)];
                    ssT_temp = s[i]*s[j];
                    xtx_temp = xtx_temp*ssT_temp;
                    ga[id(i,j,d_active.size())] = xtx_temp;
                }
        }
    double* p_xa = thrust::raw_pointer_cast(&xa[0]);
    double* p_ga = thrust::raw_pointer_cast(&ga[0]);
```

97
APPENDIX A. APPENDIX

```c
if(no_xtx)//ga = xa^T*xa;
{
    cublasDgemm(hd, CUBLAS_OP_T, CUBLAS_OP_N, d_active.size(), d_active.size(), nRow, &cc1, p_xa, nRow, p_xa, nRow, &cc0, p_ga, d_active.size());
}
c.clear();

//invga = ga\eye(size(ga,1));
thrust::device_vector<double> invga(d_active.size()*d_active.size());
    //invga
    if(ga.size()==1)
    {
        //printf("ga is scalar, invresion is simple\n");
        if(ga[0]==0)
        {
            printf("ga is zero!!!\n");
            canPrintSol = 0;
            break;
        }
        else
            invga[0] = 1/ga[0];
    }

    int batchSize = 1;
    int n = d_active.size();
    int lda = n;
    if(n<1)
    {
        printf("[\%d] the size of active is 0.\n", nIter);
        exit(EXIT_FAILURE);
    }
```

98
int *P, *INFO;
cudaMalloc<int>(&P,n * batchSize * sizeof(int));
cudaMalloc<int>(&INFO,batchSize * sizeof(int));

double* p_invga = thrust::raw_pointer_cast(&invga[0]);
const double* p_gac = thrust::raw_pointer_cast(&ga[0]);

thrust::device_vector<double*> array_pA(1);
array_pA[0] = p_ga;
double** p_pA = thrust::raw_pointer_cast(&array_pA[0]);

cublasDgetrfBatched(hd,n,p_pA,lda,P,INFO,batchSize);
int INFOh = 0;
cudaMemcpy(&INFOh,INFO,sizeof(int),cudaMemcpyDeviceToHost);
if(INFOh == n)
{
    fprintf(stderr, "Factorization Failed: Matrix is singular\n");
cudaDeviceReset();
    exit(EXIT_FAILURE);
}

thrust::device_vector<const double*> array_pAc(1);
array_pAc[0] = p_gac;
const double** p_pAc = thrust::raw_pointer_cast(&array_pAc[0]);

thrust::device_vector<double*> array_pinvA(1);
array_pinvA[0] = p_invga;
double** p_pinvA = thrust::raw_pointer_cast(&array_pinvA[0]);

cublasDgetriBatched(hd,n,p_pAc,lda,P,p_pinvA,lda,INFO,batchSize);
cudaMemcpy(&INFOh,INFO,sizeof(int),cudaMemcpyDeviceToHost);
if(INFOh != 0)
{
    fprintf(stderr, "Inversion Failed: Matrix is singular\n");
cudaDeviceReset();
exit(EXIT_FAILURE);
}
cudaFree(P), cudaFree(INFO);

// aa= sum(sum(invga))^{-1/2};% eq 2.5
double aa;
double sum_invga = 0;

thrust::device_vector<double> ones2(n, 1);
double*p_ones2 = thrust::raw_pointer_cast(&ones2[0]);
thrust::device_vector<double> sum_invga_col(n);
double*p_sum_invga_col = thrust::raw_pointer_cast(&sum_invga_col[0]);
cublasDgemv(hd, CUBLAS_OP_T, n, n, &cc1, p_invga, n, p_ones2, 1, &cc0, p_sum_invga_col, 1);
sum_invga = thrust::reduce( sum_invga_col.begin() , sum_invga_col.end() );
aa = 1/sqrt(sum_invga);

// wa= aa*sum(invga,2);% eq 2.6
thrust::device_vector<double> sum_invga_row(n);
double*p_sum_invga_row = thrust::raw_pointer_cast(&sum_invga_row[0]);
cublasDgemv(hd, CUBLAS_OP_N, n, n, &cc1, p_invga, n, p_ones2, 1, &cc0, p_sum_invga_row, 1);
trust::device_vector<double> wa(n);
double*p_wa = thrust::raw_pointer_cast(&wa[0]);
thrust::transform(sum_invga_row.begin(), sum_invga_row.end(), wa.begin(), MultipleByC(aa));

// ua= xa*wa;% eq 2.6
thrust::device_vector<double> ua(nRow);
double *p_ua = thrust::raw_pointer_cast(&ua[0]);
cublasDgemv(hd, CUBLAS_OP_N, nRow, n, &cc1, p_xa, nRow, p_wa, 1, & cc0, p_ua, 1);

// test using Eq 2.7
// test_1 = xa'T*ua;
thrust::device_vector<double> test_1(n, 0);
double *p_test_1 = thrust::raw_pointer_cast(&test_1[0]);
cublasDgemv(hd, CUBLAS_OP_T, nRow, n, &cc1, p_xa, nRow, p_ua, 1, & cc0, p_test_1, 1);

// test_2 = aa*ones(size(test_1));
thrust::device_vector<double> test_2(n, 0);
double *p_test_2 = thrust::raw_pointer_cast(&test_2[0]);
// test_1_2 = sum(sum(abs(test_1-test_2)));  
thrust::transform(test_1.begin(), test_1.end(), test_2.begin(), MinusC(aa));
    thrust::transform(test_2.begin(), test_2.end(), test_2.begin(), Abs () );
double test_1_2 = thrust::reduce( test_2.begin(), test_2.end() );

// test_3 = norm(ua) - 1;
double test_3;
const double *p_uac = thrust::raw_pointer_cast(&ua[0]);
cublasDnrm2(hd, nRow, p_uac, 1, &test_3);  
test_3 = test_3-1;
if(test_3<0) test_3 = -test_3;

if(test_1_2 > RESOLUTION_OF_LARS*100 || test_3>RESOLUTION_OF_LARS *100)
{

APPENDIX A. APPENDIX

printf("[%d]Eq 2.7 test failure.\n", nIter);
    canPrintSol = 0;
    break;
}
//a = x^T*ua;
thrust::device_vector<double> a(nCol);
double *p_a = thrust::raw_pointer_cast(&a[0]);
cublasDgemv(hd, CUBLAS_OP_T, nRow, nCol, &ccl, p_dx, nRow, p_ua, 1,
            &cc0, p_a, 1);

double c_inactive_temp;
double a_inactive_temp;
double tmp_1;
double tmp_2;
thrust::device_vector<double> tmp;
for(int k = 0; k < inactive.size(); k++)
{
    c_inactive_temp = c[inactive[k]]; //tmp_1 = (C_max-c(inactive))
        ./ (aa-a(inactive));
    a_inactive_temp = a[inactive[k]]; //tmp_2 = (C_max+c(inactive))
        ./ (aa+a(inactive));
    tmp_1 = (C_max-c_inactive_temp)/(aa-a_inactive_temp); 
    tmp_2 = (C_max+c_inactive_temp)/(aa+a_inactive_temp);
    if(tmp_1>0) tmp.push_back(tmp_1); //tmp = tmp_3(find(tmp_3>0));
    if(tmp_2>0) tmp.push_back(tmp_2);
}

double gamm;
if(tmp.empty())
    //if(tmp->empty())
    gamm = C_max/aa;
else
    { //gamma = min(tmp);
        thrust::device_vector<double>::iterator iter4 = thrust::
APPENDIX A. APPENDIX

```cpp
min_element(tmp.begin(), tmp.end());

gamma = *iter4;

thrust::device_vector<double> d(nCol, 0); // d = zeros(1, nCol);
thrust::device_vector<double> tmpp(nCol, 0); // tmp = zeros(1, nCol);
double d_temp;

double beta_temp;
thrust::device_vector<double> tmpp2;
double s_temp;

for(int k = 0; k < d_active.size(); k++)
{
    double c_active_temp2 = c[d_active[k]];
    if(c_active_temp2 >= RESOLUTION_OF_LARS) // s = sign(c(active));
        s_temp = 1;
    if(c_active_temp2 <= -RESOLUTION_OF_LARS)
        s_temp = -1;
    if(c_active_temp2 < RESOLUTION_OF_LARS && c_active_temp2 > -
        RESOLUTION_OF_LARS)
        s_temp = 0;
    d_temp = s_temp*wa[k]; // d(active) = s.*wa;
    if(d_temp==0) //if (length(find(d(active)==0))
    {
        printf("[%d]Something wrong with vector d: Eq 3.4\n", nIter);
        canPrintSol = 0;
        break;
    }
    d[d_active[k]] = d_temp;
}

for(int k = 0; k < d_active.size(); k++)
{
```

103
beta_temp = beta[d_active[k]];  

    d_temp = d[d_active[k]];  
    beta_temp = -beta_temp/d_temp;  
    tmpp[d_active[k]] = beta_temp;//tmp(active) = -1*beta(active)./d  
    (active);  

    if(beta_temp>0)  
    {  
        tmpp2.push_back(beta_temp);//tmp2 = tmp(find(tmp>0));  
        //tmpp2->push_back(beta_temp);//tmp2 = tmp(find(tmp>0));  
        //printf("%lf goes to tmpp2 in tmp2 = tmp(find(tmp>0));\n",  
            beta_temp);  
    }  
    }  
  
    drop.clear();//drop = [];
    double gamm_tilde = INF;  

    if(!tmpp2.empty())  
    {  
        thrust::device_vector<double>::iterator iter3 = thrust:::  
            min_element(tmpp2.begin(), tmpp2.end());  

        unsigned int position = iter3 - tmpp2.begin();  
        double min_tmpp2 = *iter3;  

        if(!tmpp2.empty() && gamm >= min_tmpp2)  
        {  
            gamm_tilde = min_tmpp2; //gamma_tilde = min(tmp2);  
            for(int k = 0; k < nCol; k++)//drop = find(tmp==gamma_tilde)  
            {  
                if(tmpp[k]==gamm_tilde)  
                {  

            }  

    }
drop.push_back(k);

//if(rank==10) fprintf(f_act_record, "[%d] %d added
to drop at line: 808\n", nIter, k);

} } }  

}  

}  

tmpp2.clear();  
tmpp.clear();  

double min_gamm = gamm;  
if(gamm > gamm_tilde) min_gamm = gamm_tilde;  

//const double *p_min_gamm = &min_gamm;  
mu_a_plus = mu_a;  
//double *p_mu_a_plus = thrust::raw_pointer_cast(&mu_a_plus[0]);  
//cublasDaxpy(hd, nRow, p_min_gamm, p_uac, 1, p_mu_a_plus, 1);//  
mu_a_plus = mu_a + min(gamm,gamm_tilde)*ua;  
thrust::device_vector<double> new_ua1=ua;  
thrust::transform(ua.begin(), ua.end(), new_ua1.begin(), MultipleByC(
(min_gamm));  
thrust::transform(mu_a.begin(), mu_a.end(), new_ua1.begin(),
mu_a_plus.begin(), thrust::plus<double>>());  
new_ua1.clear();  

beta_new = beta;  
//const double *p_dc = thrust::raw_pointer_cast(&d[0]);  
//double *p_beta_new = thrust::raw_pointer_cast(&beta_new[0]);  
//cublasDaxpy(hd, nCol, p_min_gamm, p_uac, 1, p_mu_a_plus, 1);//  
beta_new = beta + min(gamm,gamm_tilde)*d;  
thrust::device_vector<double> new_dl=d;  
thrust::transform(d.begin(), d.end(), new_dl.begin(), MultipleByC(
(min_gamm));
thrust::transform(beta.begin(), beta.end(), new_d1.begin(), beta_new.begin(), thrust::plus<double>())
new_d1.clear();

//active = setdiff(active, drop);
thrust::device_vector<int> new_active2(d_active.size());
thrust::device_vector<int>::iterator act_end2;
act_end2 = thrust::set_difference(d_active.begin(), d_active.end(),
    drop.begin(), drop.end(), new_active2.begin());
new_active2.erase(act_end2, new_active2.end());
d_active = new_active2;
new_active2.clear();

//inactive = setdiff(all_candidate, active);
thrust::device_vector<int>::iterator inact_end2;
inact_end2 = thrust::set_difference(all_candidate.begin(),
    all_candidate.end(), d_active.begin(), d_active.end(), inactive.begin());
inactive.erase(inact_end2, inactive.end());

for(int k = 0; k < drop.size(); k++) //beta_new(drop)=0;
{
    beta_new[drop[k]] = 0;
}

double C_max_aa;
C_max_aa = C_max/aa;
//const double * p_C_max_aa = &C_max_aa;
mu_a_OLS = mu_a;
//double *p_mu_a_OLS = thrust::raw_pointer_cast(&mu_a_OLS[0]);
//cublasDaxpy(hd, nRow, p_C_max_aa, p_uac, 1, p_mu_a_OLS, 1);//
    mu_a_OLS = mu_a + C_max/aa*ua;
thrust::device_vector<double> new_ua2=ua;
thrust::transform(ua.begin(), ua.end(), new_ua2.begin(), MultipleByC
(C_max_aa));

thrust::transform(mu_a.begin(), mu_a.end(), new_uA2.begin(),
    mu_a_OLS.begin(), thrust::plus<double>());

new_uA2.clear();

beta_OLS = beta;

    //const double *p_dc = thrust::raw_pointer_cast(&d[0]);
    //double *p_beta_OLS = thrust::raw_pointer_cast(&beta_OLS[0]);
    //cublasDaxpy(hd, nCol, p_C_max_aa, p_dc, 1, p_beta_new, 1); //
    beta_LS = beta + C_max/aa*d;

thrust::device_vector<double> new_d2=d;

thrust::transform(d.begin(), d.end(), new_d2.begin(), MultipleByC(
    C_max_aa));

thrust::transform(beta.begin(), beta.end(), new_d2.begin(), beta_OLS
    .begin(), thrust::plus<double>());

new_d2.clear();

    //MSE = sum((y-mu_a_OLS).^2)/length(y);
    thrust::device_vector<double> y_minus_mu_a(nRow);

    y_minus_mu_a = dy;
    //double *p_y_minus_mu_a = thrust::raw_pointer_cast(&y_minus_mu_a
        [0]);
    //double* p_mu_a_OLSc = thrust::raw_pointer_cast(&mu_a_OLS[0]);
    //const double c_m1 = -1;
    //cublasDaxpy(hd, nRow, &c_m1, p_mu_a_OLSc, 1, p_y_minus_mu_a, 1);
    thrust::transform(dy.begin(), dy.end(), mu_a_OLS.begin(),

        y_minus_mu_a.begin(), thrust::minus<double>());

    thrust::transform(y_minus_mu_a.begin(), y_minus_mu_a.end(),
        y_minus_mu_a.begin(), Square());

    MSE = thrust::reduce( y_minus_mu_a.begin() , y_minus_mu_a.end());

    MSE = MSE/nRow;
y_minus_mu_a.clear();
d.clear();
ua.clear();
wa.clear();
a.clear();
xa.clear();
ga.clear();
invga.clear();

//update
mu_a = mu_a_plus;
beta = beta_new;
thrust::device_vector<double> beta_backup=beta;

double d_beta_temp;
if(MSE <= MSE_th)
{

    //beta = beta(active)./sx(active);
    thrust::fill(beta.begin(), beta.end(), 0);
    for(int k = 0; k < d_active.size(); k++)
    {
        d_beta_temp = beta_backup[d_active[k]];
        d_beta_temp = d_beta_temp/ssx[d_active[k]];
        beta[d_active[k]] = d_beta_temp;
    }

    canPrintSol = 1;
    //othersStop = 1;//need to be broadcasted!
    break;
}

beta_backup.clear();

if(nIter>=MAX_ITER){printf(" Reached nIter limit!\n");}
mu_a_plus.clear();
mu_a_OLS.clear();
beta_new.clear();
beta_OLS.clear();
c.clear();
mu_a.clear();
dy.clear();
dx.clear();
//sx.clear();
ssx.clear();
if(!no_xtx) {xtx.clear();}

thrust::host_vector<double> h_beta(beta.begin(), beta.end());
thrust::host_vector<int> active(d_active.begin(), d_active.end());
/*========================END OF SOLVING================================***/
dtime = ((double)clock() - start)/CLOCKS_PER_SEC;
if(canPrintSol == 1)
{
    printf("========Solved successfully ========\n");
    //fprintf(f_record, "[%d]Runtime = %f\n", rank, exe_time);
    printf("Runtime = %f\n", dtime);
    printf("====Number of Iteration: %d====\n", nIter);
    for(int iii = 0; iii<active.size(); iii++)
    {
        printf("------active_idx:%d------beta[active_idx]:%6.8f------\n", 
                active[iii], h_beta[active[iii]]);
    }
    else
    {
        printf("========Not Solved TAT ========\n");
    }
}
timearray[iFaultCase-startFaultCaseID] = dtime;
APPENDIX A. APPENDIX

// memory clean up
b.clear();
h_beta.clear();
active.clear();

// shutdown
status = cublasDestroy(hd);

if (status != CUBLAS_STATUS_SUCCESS)
{
    fprintf(stderr, "!!!! shutdown error (A)\n");
    return EXIT_FAILURE;
}

A.clear();

double t_ave = thrust::reduce(timearray.begin(), timearray.end());
t_ave = t_ave/nTotalFaultCases;
printf("//***********========[%d] Ave. Time = %lf ========*********
", caseID,
t_ave);
timearray.clear();

/*
if(argc <= 1 || strcmp(argv[1], "-noprompt"))
{
    printf("\nPress ENTER to exit...\n");
    getchar();
}*/
    return EXIT_SUCCESS;
}
Index

$L_0$ minimization problem, [15]
$L_1$ minimization problem, [15]

Equivalent Injection, [7]

LARS, [15]
Lasso, [15]

OLS Backup Estimation, [20]
Overlapping Group Lasso, [19]

Teed lines, [10]