Unmixing 4-D Ptychographic Images
Algorithmic Approach

Huanlin Zhou(CUHK), Zhen Zhang(CUHK), and Michaela Shoffner(UTK)

August 4th, 2017

Abstract

Fast electron detectors are gaining ground in traditional high-resolution microscopy studies. In particular, 4D ptychographic datasets collected over a range of real and reciprocal space coordinates are believed to contain a wealth of information about structure and properties of materials. However, currently available data analysis methods are either too general, only allowing for analysis of simplest objects, or too reductive, effectively recreating traditional detectors from these datasets before interpretation. This project aims to explore the ways that symmetry mode analysis, the tool used to a great effect in theoretical studies of materials, can be adapted to analyze 4D datasets of materials such as multifunctional complex oxides.

Overview

In the originally provided Matlab code, the program takes in data for three models, consisting of the baseline, and two different distortions. These form the basis for our solutions, as we determine the coefficient each is multiplied by before being summed into the provided image. The program is then provided with 16 different image data sets, each with 7,225,344 values corresponding to a 192 by 192 by 14 by 14 grid (see figure 1). The coefficients, or weights, for each of the three models must be determined for every image, or unit cell. For each of the images, a linear algebra problem of the form Ax=b is set up, with A's three columns being the numbers from each of the three base models in vector form, and b having the complete data from a single unit cell as a vector. Since this is an extremely overdetermined system, the method of least squares is used to solve for the weights. Using this method gives results that are relatively close to the known answers, but the average error is still ranging from 0.0331 to 0.356, or 3.3% to 35.6% (see figure 2). A better technique, with results closer to the known values, is an important goal of this research project.

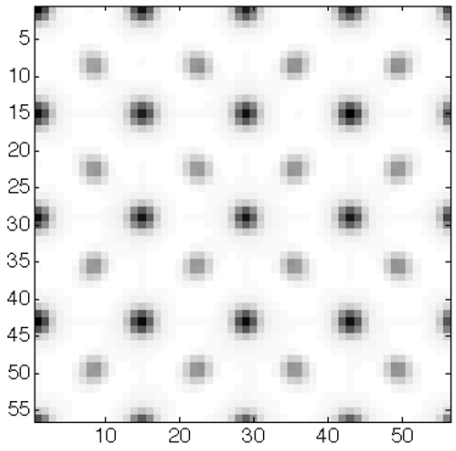

Figure 1-A representation of the image data, shown as a two-dimensional picture

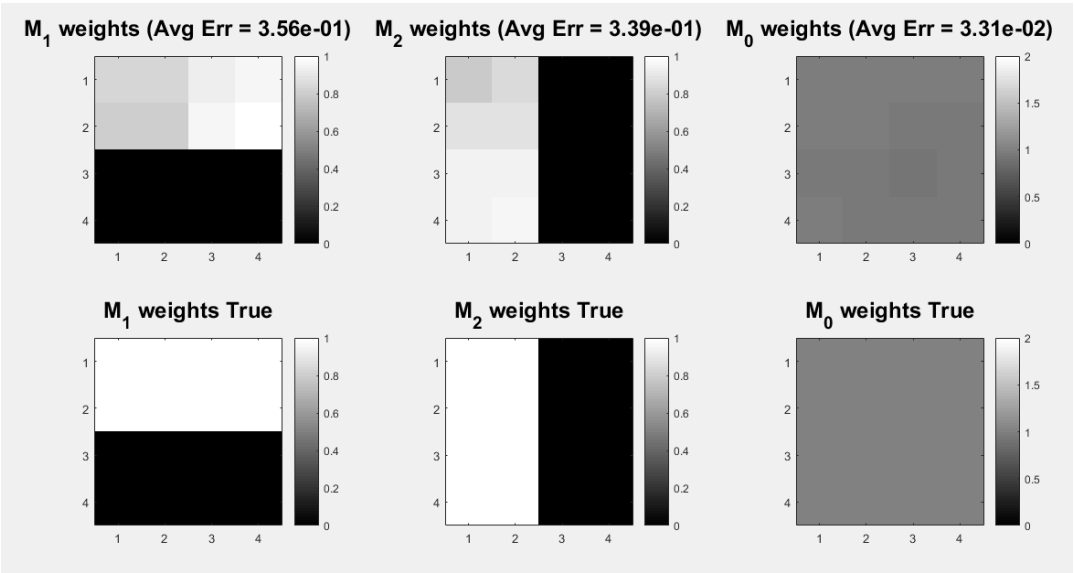

Figure 2-Initial results and error

Objectives

In the current implementation, the resulting weights are significantly different from the known values. This should primarily be because of two factors: the presence of outliers and potential nonlinear terms. Outliers will cause perceptible error when using just a simple least squares algorithm, while the current calculation assumes strictly linear terms. Our plans for improving upon the unmixing algorithm consist of two approaches. The first involves adding the gradient of each model, in both the vertical and horizontal directions, as additional columns in the model matrix, as well as averaging and condensing the data down

to a smaller set. The addition of the gradient terms should help cover any potential nonlinear terms, while the smaller data set will both help compensate for outliers and reduce the total time needed to run calculations. The second planned algorithm features the Split Bregman method, an iterative approach that would solve all of the 16 cells at the same time, but requires multiple passes to reach a reasonable answer.

The other goal for this side of the project involves converting the entire program into the C programming language. While Matlab makes interacting with matrices very easy, it sacrifices speed for a user-friendly interface. If the data processing is instead implemented using the LAPACK library in C, there should be a significant reduction in time taken by the program. Once this has been accomplished, the next step consists of changing the code to run on a GPU, using the MAGMA library. The current incarnation of the program was made to run on a basic CPU, such as a laptop, but with more processors available, far greater speed should be possible. Finally, investigations will be made into various options for making the code parallel, to further increase speed. Formats to be tried include Message Passing Interface(MPI), OpenMP, and ScaLAPACK.

Least Squares Improvement

Since the true weights are piecewise constant, we want the calculated weights to be piecewise constant as well. To force the calculated weights to be more piecewise constant, we add an L1-regularization: |grad(w)|, which is defined by:

$$|\text{grad(w)}| = \sum_{i=1}^{3} |\text{w(i} + 1, \text{j)} - \text{w(i, j)}| + \sum_{j=1}^{3} |\text{w(i, j} + 1) - \text{w(i, j)}|$$

It acts like the total variation of the matrix w. We want the error as well as this total variation to be as small as possible, so we formulate an L1-regularized problem:

$$\min |\text{grad(w)}| + \text{m}||\text{Aw} - \text{x}||_2{}^2$$

Here m is an optimization parameter whose value depends on how small we want the gradient or how small we want the error. For example, if we choose a larger m, we attach more importance to the L2-error. To avoid possible contradiction between different sets of solutions, we sum up the gradients of all 3 modes and the error of all 16 unit cells to get this single expression:

$$\sum_1^3 |\text{grad(w)}| + \text{m} \sum_1^{16} ||\text{Aw} - \text{x}||_2{}^2.$$

By minimizing it, we can solve for all of the 48 unknown weights together.

To solve the L1-regularized problem, we apply a Split Bregman method. The standard model is:

$$\min |\Phi(\text{u})| + \text{H(u)}$$

Here | | is the L1-norm. Since we want to solve for 48 unknowns, here u is a 48-by-1 column vector.

Since we have 4x4 blocks and each consists of 3 modes, we can represent the weights as:

$$\begin{bmatrix} \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma \\ \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma \\ \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma \\ \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma & \alpha, \beta, \gamma \end{bmatrix}$$

Where α, β, and γ represent the weights for mode1s 1, 2, and 0. Then, taking the entries row by row, we get this 48-by-1 vector, u.

We let E_1, E_2 be two 36x48 matrices for gradient calculation:

$$E_1 = \begin{bmatrix} 1\ 0\ 0 - 1\ 0\ 0\ 0, \dots, 0 \\ 0\ 1\ 0\ 0 - 1\ 0\ 0, \dots, 0 \\ \dots\ \dots \\ \dots\ \dots \\ 0\ 0\ 0, \dots\ 0\ 1\ 0\ 0 - 1 \end{bmatrix}$$

E_1 is for w(i, j)-w(i, j+1). For example, the first row of E_1*u is (u1-u4), which is w(1,1)-w(1,2). Similarly, E_2 is for w(i, j)-w(i+1, j). Then we get:

$$\Phi_1(u)=E_1*u,\ \Phi 2(u)=E_2*u$$

A is diagonal in block sense:

$$A = \begin{bmatrix} [M1\ M2\ M0] & \cdots & 0 \\ \vdots & [M1\ M2\ M0] & \vdots \\ 0 & \cdots & [M1\ M2\ M0] \end{bmatrix}$$

It consists of 16 identical diagonal blocks [M1 M2 M0]. x is a column vector containing all data in 16 unit cells:

$$x = \begin{pmatrix} I_1 \\ I_2 \\ \vdots \\ \vdots \\ I_{16} \end{pmatrix}$$

We then get:

$$H(u)=m||Au-x||_2^2$$

The Split Bregman method uses d_1, d_2 to approximate E_1*u and E_2*u, so it splits the L1 and L2 parts. Now the problem becomes finding the u, d_1, and d_2 that minimize:

$$|d_1|+|d_2|+H(u)+\lambda||E_1u-d_1||_2^2+\lambda||E_2u-d_2||_2^2$$

Then we can solve it iteratively as follows:

To choose appropriate $u^0, d_1^0, d_2^0, b_1^0, b_2^0$, we let u^0 be the least squares result, and all others be zeros.

Then for k=0, 1,…, N:

$$u^{k+1}=\min_u(H(u) + \lambda|| d_1^k - E_1u - b_1^k\|_2^2 + \lambda\|d_2^k - E_2u - b_2^k\|_2^2);$$

$$d_1^{k+1} = \min_d(|d| + \lambda||d - E_1u^{k+1} - b_1||_2^2);$$

$$d_2^{k+1} = \min_d(|d|+\lambda||d - E_2u^{k+1} - b_2||_2^2);$$

$$b_1^{k+1} = b_1^k + E_1u^{k+1} - d_1^{k+1};$$

$$b_2^{k+1} = b_2^k + E_2u^{k+1} - d_2^{k+1};$$

Since $H(u) = m||Au-x||_2^2 = m(Au-x)^T*(Au-x)$;

$$\lambda||d_i^k-E_iu-b_i^k||_2^2 = \lambda(E_iu- d_i^k + b_i^k)^T*(E_iu- d_i^k + b_i^k), i=1,2.$$

It is easy to calculate its gradient, and, by using the first derivative test, the minimum point is

$$u^{k+1}=(mA^TA+\lambda E_1^TE_1+\lambda E_2^TE_2)^{-1}(mA^Tu+\lambda E_1^T(d_1^k - b_1^k)+\lambda E_2^T(d_2^k - b_2^k))$$

Since A^TA, $E_1^TE_1$, $E_2^TE_2$, and A^Tx remain unchanged in each iteration, we can calculate them before the iteration starts and keep the values, which saves a lot of time.

d_i^{k+1} can be calculated by

$$(d_i^{k+1})_j = \text{shrink}\ (\Phi_i(u)_j+(b_i^k)_j, 1/\lambda) = \frac{\Phi i(u)j+(b_i^k)j}{|\Phi i(u)j+(b_i^k)j|} * \max(\left|\Phi_i(u)_j + (b_i^k)_j\right| - 1/\lambda, 0), i=1,2$$

(reference: The Split Bregman method for L1-regularized problem).

To find the correct model:

The resulting image might be linear terms plus some combination of the gradients of the 3 modes (suggested by Dr. Archibald). To simplify the data, we take the average over each of the 192x192 pixels, that is, M=squeeze(mean(mean(M))), to make each model 14x14. And here each mode has 2 gradient matrices: Gx and Gy, also of size 14x14. Gx is the gradient in the x (row) direction:

$$Gx(:,j) = M(:, j+1) – M(:, j) \quad \text{for } j < 14, \text{ and}$$

$$Gx(:,14) = M(:, 1) – M(:, 14) \quad \text{for } j = 14$$

Gy is the gradient in the y (column) direction:

$$Gy(i, :) = M(i+1, :) – M(i, :) \quad \text{for } i < 14, \text{ and}$$

$$Gy(14, :) = M(1, :) – M(14, :) \quad \text{for } i = 14$$

We assume

$$x=\alpha M1+\beta M2+\gamma M0+a_1G1x+b_1G2x+c_1G0x+a_2G1y+b_2G2y+c_2G0y$$

By the least squares method, the resulting weights (α, β, γ) are much closer to true weights; previously, the total absolute difference was 11.6405, now it is 2.9635.

If we apply the above Split Bregman method to the model with the gradients added, then u is of length 144 (3 modes * 16 unit cells * (mode itself + gradient in x direction + gradient in y direction)). u_1 to u_{48} are the weights of the modes themselves, u_{49} to u_{96} are the weights of the gradients in the x direction, and u_{97} to u_{144} are the weights of the gradients in the y direction. E_1 and E_2 are 36-by-144. $E_i(:, 1:48)$ is the same as above and all other entries are zeroes, so that E_i*u is still the gradient of u_1 through u_{48}. A is size 3,136-by-144, and is built as:

$$A = \begin{bmatrix} [M1\ M2\ M0] & [G1x\ G2x\ G0x] & & [G1y\ G2y\ G0y] \\ & \cdots & & \\ & & \cdots & \\ [M1\ M2\ M0] & [G1x\ G2x\ G0x] & & [G1y\ G2y\ G0y] \end{bmatrix}$$

So that the i-th block of A*u will be:

$$\alpha*M1+\beta*M2+\gamma*M0+a_1*G1x+b_1*G2x+c_1*G0x+a_2*G1y+b_2*G2y+c_2*G0y$$

for the i-th unit cell, i=1,2,…,16. Now we have A, Φ_1 and Φ_2, the rest is the same as above.

The result depends on the choice of m. So far, $m = 2.0917*10^{13}$ has given the best result, with the converging value of total difference being about 2.8833, slightly better than the least squares method with gradient.

See appendix for the Matlab code that implements both the least squares plus gradient method and the Split Bregman method.

Tables 1 through 3 give experimental results from the different algorithms.

Table 1-model 1 weights

Unit Cell #	Desired Value	Basic Least Squares	Least Squares with Gradient	Split Bregman with Gradient
1	1.0000	0.8284	0.7063	0.7333
2	1.0000	0.8349	0.9238	0.9199
3	1.0000	0.9280	1.0107	0.9878
4	1.0000	0.9544	1.0271	1.0000
5	1.0000	0.8117	0.8202	0.7971
6	1.0000	0.8117	0.8443	0.8443
7	1.0000	0.9676	0.9142	0.9015
8	1.0000	0.9881	0.9366	0.9101
9	-1.0000	-0.4186	-0.9917	-0.9934
10	-1.0000	-0.4692	-1.0185	-0.9937
11	-1.0000	-0.3302	-0.8604	-0.9095
12	-1.0000	-0.2990	-0.9759	-0.9446
13	-1.0000	-0.4570	-1.1031	-1.0663
14	-1.0000	-0.5001	-1.0838	-1.0514
15	-1.0000	-0.3538	-0.9573	-0.9804
16	-1.0000	-0.3590	-1.0386	-1.0008
TOTAL DIFF		5.6886	1.3511	1.2027
AVG ERROR		0.3560	0.0844	0.0753

Table 2-model 2 weights

Unit Cell #	Desired Value	Basic Least Squares	Least Squares with Gradient	Split Bregman with Gradient
1	1.0000	0.7945	0.7436	0.7697
2	1.0000	0.8472	0.9840	0.9354
3	-1.0000	-0.4802	-0.9208	-0.9070
4	-1.0000	-0.4999	-0.8786	-0.8958
5	1.0000	0.8798	0.8228	0.8454
6	1.0000	0.8754	0.8844	0.8844
7	-1.0000	-0.4627	-0.9958	-0.9290
8	-1.0000	-0.4921	-1.0007	-0.9450
9	1.0000	0.9440	0.9605	0.9623
10	1.0000	0.9474	0.9840	0.9596
11	-1.0000	-0.3569	-0.9774	-0.9211
12	-1.0000	-0.4003	-0.7466	-0.8381
13	1.0000	0.9379	0.9905	0.9809
14	1.0000	0.9591	1.1163	1.0582
15	-1.0000	-0.3422	-0.8607	-0.8381
16	-1.0000	-0.3582	-0.7989	-0.8130
TOTAL DIFF		5.4221	1.6083	1.6712
AVG ERROR		0.3390	0.0980	0.1040

Table 3-base model weights

Unit Cell #	Desired Value	Basic Least Squares	Least Squares with Gradient	Split Bregman with Gradient
1	1.0000	0.9950	1.0004	1.0004
2	1.0000	0.9924	1.0002	1.0002
3	1.0000	0.9700	1.0004	1.0004
4	1.0000	0.9712	1.0004	1.0004
5	1.0000	0.9927	1.0000	1.0000
6	1.0000	0.9882	1.0000	1.0000
7	1.0000	0.9652	1.0003	1.0003
8	1.0000	0.9645	0.9997	0.9997
9	1.0000	0.9678	1.0000	1.0000
10	1.0000	0.9631	1.0003	1.0003
11	1.0000	0.9320	1.0004	1.0004
12	1.0000	0.9467	1.0003	1.0003
13	1.0000	0.9716	0.9997	0.9997
14	1.0000	0.9660	0.9995	0.9995
15	1.0000	0.9409	1.0001	1.0001
16	1.0000	0.9426	0.9998	0.9998
TOTAL DIFF		0.5304	0.004199	0.004130
AVG ERROR		0.0331	0.000258	0.000258

C code, using a GPU, and working in parallel

While the difference in speeds between Matlab and C may be inconsequential for smaller problems, since this computation can involve matrices with over seven million rows, the speed differential is far more dramatic. Thus, it should be more efficient to convert the entire program to C code.

As a starting point to be improved upon, the Matlab version of the basic least squares runs about 35 seconds.

Because Matlab is set up to make matrix manipulation as intuitive as possible, it takes a bit of effort to determine the equivalent statements in C. For the first iteration, this is where LAPACK comes in, a C library which supplies functions made to work with large matrices. While it can be a bit harder to follow, and care must be taken with memory allocation, LAPACK allows for a much simpler transition than if we had to create it all from the beginning. The other challenge lay in obtaining the image data. This data is initially supplied by four DM3 files, the format used by the machines that generated the data. These files are not simple to read from, and the original code contained a convoluted function just to interpret the data files. Since it appeared to be very difficult to replicate in C, insofar as our knowledge extends, we elected to take the simpler route and simply wrote a Matlab program that used the given function to read in the data, then had it print the matrices to a simple text file, which could be easily, albeit slowly, read into the C variables.

With all that formatting out of the way, it merely took the correct use of the functions dgels and memcpy to get practically identical results to the original program. While we are certain it could be more elegantly implemented, this was a satisfactory first attempt, though the time needed to run was barely changed, coming in at 33 seconds. The biggest problem at this stage would be that so much time is taken to read in the data that the quicker calculations are largely off-set. This was solved by changing the data files from ascii text into binary, allowing large blocks of data to be read far more quickly. The final result has a run time of about 7 seconds.

Both the least squares plus gradient algorithm and the Split Bregman method were likewise implemented in C using LAPACK and binary files, with both having a run time of approximately 0.75 seconds.

Now having a baseline to start from, we next considered how to adapt it to run on a GPU. While changing it to C resulted in a significant reduction in run time, using a GPU should hopefully see additional decreases. In order to achieve this, we turned to the MAGMA C library, an implementation of LAPACK meant to run on a GPU, for extremely large matrices. Since MAGMA was made rather recently, it has much less in the way of online documentation, and what exists is harder to parse for an inexperienced programmer. Because of this, this step took significant research for associated literature and some trial and error efforts to achieve results. Once a working version of MAGMA code was completed, it had an unexpected result. Likely because of the time needed to move all the data from the CPU to the GPU, the code using MAGMA takes far longer than the other implementations, coming in at around 50 seconds.

The least squares plus gradient version was similarly adapted, likewise with a drop in speed, though not as dramatic, running about 4 seconds.

The final goal consisted of writing versions of the program to run in parallel on multiple CPU processors, still using LAPACK. Both Message Passing Interface (MPI) and OpenMP were explored to see what further improvements could be made on running time.

When beginning work with MPI, we started with the least squares with gradient code, since it was giving satisfactory results, but it was still straightforward to determine how to adapt it. Since there are 16 unit cells, and their calculations are independent of each other, they can easily be done in parallel. The first version of MPI code was quite simple: set the number of processes to be 16, and have process 0 read and simplify

all the data. Each process calls MPI_Bcast to broadcast A to all 16 processes, and calls MPI_Scatter to divide x into 16 equal-sized parts (one unit cell to each processor). After that, each process, having a copy of A and the data of one of the 16 unit cells, works through the least squares calculation and finds the weights of the 3 modes. After finishing the calculations, each process calls MPI_Gather to send the calculated weights back to process 0, after which process 0 prints out all the results.

Since some amount of time is used for reading in all the data, we also considered reading data in parallel: dividing x into 16 separate data files, each process reading in one file and doing the simplification, with the first 3 processes also reading in and simplifying the data of the three standard modes. Then processes 1 to 15 send the simplified data to process 0, and the rest is completed identically to before.

The MPI code for the least squares plus gradient algorithm performed reasonably well, if not as well as we'd hoped. The second version tends to outperform the first, with an average reduction of about 0.9 seconds, but both can be very volatile in their speed. Additionally, with the most consistent time elapsed for the second version being around 2.8 seconds, it still lags behind the basic LAPACK implementation, though even the first version was at least faster than MAGMA or Matlab.

One final variation was made, upon realizing that it was needless to have each process send the simplified data file to process 0, only to re-distribute it back to all the processes. After removing this step, and having each process act upon the data it just read and simplified, the time was drastically reduced, now running between 0.7 and 1.4 seconds.

When MPI was used for the basic least squares algorithm (only version one), it also performed poorly, with the running time clocking in at approximately 53 seconds, even longer than the MAGMA implementation.

OpenMP, implemented for the same two algorithms, fared better, with its best speeds tending to run just under or just over the speed of the LAPACK implementations. This more or less equivalent running time is still not the significant improvement we'd hoped for, but at least it wasn't significantly worse like the MPI versions. Unexpectedly, it runs faster on average when fewer threads are used, with its peak speed in the least squares plus gradient version actually being noticeably better than the standard LAPACK implementation, coming in at 0.665 seconds. Unfortunately, this implementation had the most volatile running times so far, with vastly different times reported among each thread count tested (1, 2, 4, 8, 16); this would need further testing before good conclusions can be drawn.

As for the Split Bregman method, since all 16 cells are calculated simultaneously already, and each iteration needs the result from the previous, it cannot be directly performed in parallel. However, inside each iteration, we must solve a 144x144 linear system, and this can be done in parallel using ScaLAPACK, a parallel version of LAPACK.

ScaLAPACK chooses 2-D block cyclic data distribution to optimize BLAS3 (matrix-matrix) operation. It is not as straightforward as LAPACK. We need to distribute the data to each process in a process grid manually, in a cyclic manner. Then call the ScaLAPACK subroutine pdgesv. In the distribution of the data, we set the process grid to be 4x4, that is, there are 16 processes. We divide the 144x144 matrix into 2x2 small blocks. Since we have 4x4 processes and each block is 2x2, each cycle should be 8x8, and there are totally 18x18 cycles. In a cycle, each process takes its 2x2 block. Then each process has a 36x36 submatrix. For vector b, it is only divided in column direction, in the same way as A. And only processes in column 0 has their part of b, which is a 36x1 vector. The local b for other processes is a zero vector.

After calling pdgesv, only processes in column 0 have their part of the solution, in the same location as their part of b. For example, if process (0,0) has b1 and b2, then after the routine, it will have u1 and u2.

Since in the next step, we need E_1u and E_2u where u is the complete solution, we have to gather different parts of the solution together. To do this, we initialize u for each process to be a 144x1 zero vector, then put the parts of the solution into their corresponding location in u, leaving all other entries zero. Then we can simply sum up all 16 u's to get the complete solution by calling blacs routine dgsum2d, which is the routine that performs element-wise summation over all processes.

The time for running this ScaLAPACK code is even more inconsistent than OpenMP, ranging from 30 seconds to over 6 minutes, usually taking much longer than any previous version. That might because we let each process read in the full set of data, and that takes lots of memory. Also, ScaLAPACK is designed for solving very large system, however, our system is only 144x144, too small for ScaLAPACK to show its advantage. Therefore, what can be done next might be letting process 0 read and simplify all the data and send other processes their own part, which can reduce the memory usage. We may also let each process read and simplify one part of the data and generate their own local data, just as what we have done with MPI. We simply did not have the time to continue working on this program during our allotted time.

See tables 4 through 6 for a complete list of running times of the three algorithms in various forms.

Basic least squares - Program version	Peak Running time
Original code, run from Matlab on a laptop, using DM3 files	~ 35 seconds
LAPACK implementation on CPU, run from Bridges system, using text files	~ 33 seconds
LAPACK implementation on CPU, run from Bridges system, using binary files	~ 7 seconds
MAGMA implementation on GPU, run from Bridges system, using text files	~ 87 to 98 seconds
MAGMA implementation on GPU, run from Bridges system, using binary files	~ 50 seconds
MPI implementation mk 1, run from Bridges system, using binary files	~ 53 seconds
OpenMP implementation, run from Bridges system, using binary files	~ 7.4 to 26.3 seconds

Table 4-program running times for versions of the basic least squares algorithm, all generating the same results

Least squares plus gradient - Program version	Peak Running time
First version, in Matlab, run from Bridges system, using DM3 files	~ 21 seconds
LAPACK implementation on CPU, run from Bridges system, using binary files	~ 0.75 seconds
MAGMA implementation on GPU, run from Bridges system, using binary files	~ 4 seconds
MPI implementation mk 1, run from Bridges system, using binary files	~ 3.7 seconds
MPI implementation mk 2, run from Bridges system, using binary files	~ 2.8 seconds
MPI implementation mk 3, run from Bridges system, using binary files	~ 0.7 to 1.4 seconds
OpenMP implementation, run from Bridges system, using binary files	0.66 to 2.99 seconds

Table 5-program running times for versions of the least squares plus gradient algorithm, all generating the same results

Split Bregman method - Program version	Peak Running time
First version, in Matlab, run from Bridges system, using DM3 files	~ 22 seconds
LAPACK implementation on CPU, run from Bridges system, using binary files	~ 0.76 seconds
MPI implementation, run from Bridges system, using binary files	~ 1.5 seconds
ScaLAPACK implementation on CPU, run from Bridges system, using binary files	~ 30 seconds

Table 6-program running times for versions of the Split Bregman algorithm, all generating the same results

Next Steps

Should this project be continued, recommended next steps for the algorithmic approach include the following: 1) further investigation into why OpenMP is faster the fewer threads are requested and how to make it run more as expected, 2) continuing to work with ScaLAPACK to see if it could be a viable alternative to our other parallel techniques, and above all, 3) further testing with larger pools of data, to determine how these results expand beyond our limited example group.

Acknowledgements

Thanks to our mentors: Dr. Rick Archibald(ORNL), Dr. Azzam Haidar(UTK), Dr. Stanimire Tomov(UTK), and Dr. Kwai Wong(UTK). We've learned so much this summer, thanks to you.

This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

This project was sponsored by Oak Ridge National Laboratory, the Joint Institute for Computational Sciences, the University of Tennessee, Knoxville, and the Chinese University of Hong Kong.

This project was made possible by support from the National Science Foundation.

Works cited

Matlab documentation index. Mathworks Inc., www.mathworks.com/help/matlab/index.html. Accessed periodically from June 3rd, 2017 through June 21st, 2017.

Intel Software Developer Zone. Intel, https://software.intel.com/en-us. Accessed periodically throughout June and July 2017.

Chrzęszczyk, Andrzej, and Jakub Chrzęszczyk. "Matrix computations on the GPU CUBLAS and MAGMA by example." August 2013. PDF. Accessed June 27th, 2017.

Goldstein, Tom, and Stanley Osher. "The Split Bregman Method for L1-Regularized Problems." *Siam Journal on Imaging Sciences*, vol. 2, no. 2, 2009, pp. 323-343.

LAPACK: Linear Algebra PACKage. 3.7.1, University of Tennessee, University of California Berkeley, University of Colorado Denver, NAG Ltd., www.netlib.org/lapack/explore-html/index.html. Accessed periodically from June 10th through June 30th.

Towns, John, et al. "XSEDE: Accelerating Scientific Discovery", *Computing in Science & Engineering*, vol. 16, no. 5, pp. 62-74, Sept.-Oct. 2014, doi: 10.1109/MCSE.2014.80

Appendix

New algorithms – Matlab code

```
%Least squares with gradient, final version

clear;

addpath('D:\');


full_data = DM3Import('STO_Modes_even.dm3');


[Nx Ny Nz] = size(full_data.image_data);

N_full = sqrt(Nz);


full_data.image_data = reshape(full_data.image_data,[Nx Ny N_full
N_full]);


image_full_I = zeros(Nx*N_full,Ny*N_full);

for jx = 1:N_full

    for jy = 1:N_full

        image_full_I((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
full_data.image_data(:,:,jx,jy);

    end

end


Model1 = DM3Import('STO_Mode1_14x14_cbeds.dm3');


[Nx Ny Nz] = size(Model1.image_data);

N_model = sqrt(Nz);


Model1.image_data = reshape(Model1.image_data,[Nx Ny N_model
N_model]);
```

```
Model2 = DM3Import('STO_Mode2_14x14_cbeds.dm3');


[Nx Ny Nz] = size(Model2.image_data);


Model2.image_data = reshape(Model2.image_data,[Nx Ny sqrt(Nz)
sqrt(Nz)]);


image_full_M1 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));

for jx = 1:sqrt(Nz)

    for jy = 1:sqrt(Nz)

        image_full_M1((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model1.image_data(:,:,jx,jy);

    end

end


image_full_M2 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));

for jx = 1:sqrt(Nz)

    for jy = 1:sqrt(Nz)

        image_full_M2((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model2.image_data(:,:,jx,jy);

    end

end



Model_base = DM3Import('STO_mode0_14x14_cbeds.dm3');

[Nx Ny Nz] = size(Model_base.image_data);

N_model = sqrt(Nz);

Model_base.image_data = reshape(Model_base.image_data,[Nx Ny N_model
N_model]);


image_full_M0 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));

for jx = 1:sqrt(Nz)
```

```
    for jy = 1:sqrt(Nz)
        image_full_M0((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model_base.image_data(:,:,jx,jy);
    end
end
M1=squeeze(mean(mean(Model1.image_data)));
M2=squeeze(mean(mean(Model2.image_data)));
M0=squeeze(mean(mean(Model_base.image_data)));
 M1=M1-M0;
 M2=M2-M0;
 l=196;



        g1x=zeros(14);
        g1y=zeros(14);
        g2x=zeros(14);
        g2y=zeros(14);
        g0x=zeros(14);
        g0y=zeros(14);
 for i=1:14
    for j=1:14
      if j<14
         g1x(i,j)=M1(i,j+1)-M1(i,j);
         g2x(i,j)=M2(i,j+1)-M2(i,j);
         g0x(i,j)=M0(i,j+1)-M0(i,j);
      else
          g1x(i,j)=M1(i,1)-M1(i,j);
          g2x(i,j)=M2(i,1)-M2(i,j);
          g0x(i,j)=M0(i,1)-M0(i,j);
      end
    end
```

```
 end

 for i=1:14
   for j=1:14
      if i<14
         g1y(i,j)=M1(i+1,j)-M1(i,j);
         g2y(i,j)=M2(i+1,j)-M2(i,j);
         g0y(i,j)=M0(i+1,j)-M0(i,j);
      else
          g1y(i,j)=M1(1,j)-M1(i,j);
           g2y(i,j)=M2(1,j)-M2(i,j);
          g0y(i,j)=M0(1,j)-M0(i,j);
      end
   end
 end
  w_Model1_prim = zeros(4);
w_Model2_prim = zeros(4);
w_Model_base_prim = zeros(4);
err_prim = zeros(4);

w_Model1_true = zeros(4);
w_Model1_true(:,1:2) = 1;
w_Model1_true(:,3:4) = -1;
w_Model2_true = zeros(4);
w_Model2_true(1:2,:) = 1;
w_Model2_true(3:4,:) = -1;
w_Model_base_true = ones(4);
A=[M1(:),M2(:),M0(:),g1x(:),g2x(:),g0x(:),g1y(:),g2y(:),g0y(:)];

for jx = 1:4
```

```
    for jy = 1:4
    x = full_data.image_data(:,:,(1:14)+14*(jy-1),(1:14)+14*(jx-1));
    x=squeeze(mean(mean(x)));
    w=A\x(:);
    w_Model1_prim(jx,jy)=w(1);
    w_Model2_prim(jx,jy)=w(2);
    w_Model_base_prim(jx,jy)=w(3);
    end
end
total_diff=sum(abs(w_Model1_prim(:)-
w_Model1_true(:)))+sum(abs(w_Model2_prim(:)...
    -w_Model2_true(:)))+sum(abs(w_Model_base_prim(:)-
w_Model_base_true(:)));
```

```
%Split Bregman method, final version
clear;
addpath('/pylon2/ac5610p/huanlin');

L=input('lambda=');
N=input('number of iteration=');
M=input('coeff of ||Au-x||_2=');

full_data = DM3Import('STO_Modes_even.dm3');

[Nx Ny Nz] = size(full_data.image_data);
N_full = sqrt(Nz);

full_data.image_data = reshape(full_data.image_data,[Nx Ny N_full
N_full]);

image_full_I = zeros(Nx*N_full,Ny*N_full);
for jx = 1:N_full
    for jy = 1:N_full
        image_full_I((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
full_data.image_data(:,:,jx,jy);
    end
end

Model1 = DM3Import('STO_Mode1_14x14_cbeds.dm3');

[Nx Ny Nz] = size(Model1.image_data);
N_model = sqrt(Nz);

Model1.image_data = reshape(Model1.image_data,[Nx Ny N_model
N_model]);
```

```
Model2 = DM3Import('STO_Mode2_14x14_cbeds.dm3');


[Nx Ny Nz] = size(Model2.image_data);


Model2.image_data = reshape(Model2.image_data,[Nx Ny sqrt(Nz)
sqrt(Nz)]);


image_full_M1 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));

for jx = 1:sqrt(Nz)

    for jy = 1:sqrt(Nz)

        image_full_M1((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model1.image_data(:,:,jx,jy);

    end

end


image_full_M2 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));

for jx = 1:sqrt(Nz)

    for jy = 1:sqrt(Nz)

        image_full_M2((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model2.image_data(:,:,jx,jy);

    end

end



Model_base = DM3Import('STO_mode0_14x14_cbeds.dm3');

[Nx Ny Nz] = size(Model_base.image_data);

N_model = sqrt(Nz);

Model_base.image_data = reshape(Model_base.image_data,[Nx Ny N_model
N_model]);


image_full_M0 = zeros(Nx*sqrt(Nz),Ny*sqrt(Nz));
```

```
for jx = 1:sqrt(Nz)
    for jy = 1:sqrt(Nz)
        image_full_M0((1:Nx)+Nx*(jx-1),(1:Ny)+Ny*(jy-1)) =
Model_base.image_data(:,:,jx,jy);
    end
end
M1=squeeze(mean(mean(Model1.image_data)));
M2=squeeze(mean(mean(Model2.image_data)));
M0=squeeze(mean(mean(Model_base.image_data)));
 M1=M1-M0;
 M2=M2-M0;
 l=196;
 x=zeros(16*196,1);
 for jx = 1:4
    for jy = 1:4
        x_tmp = full_data.image_data(:,:,(1:14)+14*(jy-
1),(1:14)+14*(jx-1));
        x_tmp=squeeze(mean(mean(x_tmp)));
      x(((4*(jx-1)+jy-1)*l+1):(4*(jx-1)+jy)*l)=x_tmp(:);
    end
 end


        g1x=zeros(14);
        g1y=zeros(14);
        g2x=zeros(14);
        g2y=zeros(14);
        g0x=zeros(14);
        g0y=zeros(14);
 for i=1:14
    for j=1:14
      if j<14
```

```
        g1x(i,j)=M1(i,j+1)-M1(i,j);
        g2x(i,j)=M2(i,j+1)-M2(i,j);
        g0x(i,j)=M0(i,j+1)-M0(i,j);
      else
         g1x(i,j)=M1(i,1)-M1(i,j);
         g2x(i,j)=M2(i,1)-M2(i,j);
         g0x(i,j)=M0(i,1)-M0(i,j);
     end
   end
end

for i=1:14
  for j=1:14
     if i<14
        g1y(i,j)=M1(i+1,j)-M1(i,j);
        g2y(i,j)=M2(i+1,j)-M2(i,j);
        g0y(i,j)=M0(i+1,j)-M0(i,j);
     else
         g1y(i,j)=M1(1,j)-M1(i,j);
         g2y(i,j)=M2(1,j)-M2(i,j);
         g0y(i,j)=M0(1,j)-M0(i,j);
    end
  end
end
w_Model1_true = zeros(4);
w_Model1_true(:,1:2) = 1;
w_Model1_true(:,3:4) = -1;
w_Model2_true = zeros(4);
w_Model2_true(1:2,:) = 1;
w_Model2_true(3:4,:) = -1;
```

```
w_Model_base_true = ones(4);

E1_tmp=zeros(36,48);
for k=1:4
  for i=1:9
    E1_tmp(i+9*(k-1),i+9*(k-1)+3*(k-1))=1;
    E1_tmp(i+9*(k-1),i+9*(k-1)+3*k)=-1;
  end
end
E1=zeros(36,144);
E1(:,1:48)=E1_tmp;

E2_tmp=zeros(36,48);
for i=1:36
  E2_tmp(i,i)=1;
  E2_tmp(i,i+12)=-1;
end
E2=zeros(36,144);
E2(:,1:48)=E2_tmp;

A_unit=[M1(:),M2(:),M0(:)];
Gx_unit=[g1x(:),g2x(:),g0x(:)];
Gy_unit=[g1y(:),g2y(:),g0y(:)];

err_sum_origin=0;
l=196;
A=zeros(l*16,48*3);

  for jx = 1:4
    for jy = 1:4
```

```
      A(((4*(jx-1)+jy-1)*l+1):(4*(jx-1)+jy)*l,(3*(4*(jx-1)+jy-
1)+1):3*(4*(jx-1)+jy))=A_unit;

      A(((4*(jx-1)+jy-1)*l+1):(4*(jx-1)+jy)*l,(3*(4*(jx-1)+jy-
1)+1+48):3*(4*(jx-1)+jy)+48)...

     =Gx_unit;

      A(((4*(jx-1)+jy-1)*l+1):(4*(jx-1)+jy)*l,(3*(4*(jx-1)+jy-
1)+1+96):3*(4*(jx-1)+jy)+96)...

     =Gy_unit;

    end

 end


u=zeros(48*3,1);

for i=1:4

   for j=1:4

       u(12*(i-1)+3*(j-1)+1)=w_Model1_true(i,j);

       u(12*(i-1)+3*(j-1)+2)=w_Model2_true(i,j);

       u(12*(i-1)+3*(j-1)+3)=w_Model_base_true(i,j);

   end

end

u_true=u;


% Split bregman iteration.

d1=E1*u; d2=E2*u; b1=zeros(36,1); b2=zeros(36,1);

AtransA=A'*A;

Atransx=A'*x;

for n=1:N

   disp(['n=',num2str(n)]);


u=(M*AtransA+(L/2)*(E1'*E1)+(L/2)*(E2'*E2))\(M*Atransx+(L/2)*E1'*(d1-
b1)+(L/2)*E2'*(d2-b2));

     tmp1=E1*u;

     tmp2=E2*u;
```

```
   for j=1:36
       xj1=tmp1(j)+b1(j);
       d1(j)=(xj1/abs(xj1))*max(abs(xj1)-1/L,0);
       xj2=tmp2(j)+b2(j);
       d2(j)=(xj2/abs(xj2))*max(abs(xj2)-1/L,0);
    end
  b1=b1+E1*u-d1;
  b2=b2+E2*u-d2;
end

w_Model1_prim = zeros(4);
w_Model2_prim = zeros(4);
w_Model_base_prim = zeros(4);

for i=1:4
  for j=1:4
     w_Model1_prim(i,j)= u(12*(i-1)+3*(j-1)+1);
     w_Model2_prim(i,j)= u(12*(i-1)+3*(j-1)+2);
    w_Model_base_prim(i,j)= u(12*(i-1)+3*(j-1)+3);
end
end
total_diff=sum(abs(w_Model1_prim(:)-
w_Model1_true(:)))+sum(abs(w_Model2_prim(:)...
    -w_Model2_true(:)))+sum(abs(w_Model_base_prim(:)-
w_Model_base_true(:)));
```