REORDERED SUBSETS RECONSTRUCTION OF PROTON COMPUTED TOMOGRAPHY

A Project

Presented to the

Faculty of

California State University,

San Bernardino

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

---

by

Wenzhe Xue

September 2010

REORDERED SUBSETS RECONSTRUCTION OF PROTON COMPUTED TOMOGRAPHY

A Project

Presented to the

Faculty of

California State University,

San Bernardino

---

by

Wenzhe Xue

September 2010

Approved by:

Haiyan Qiao, Advisor, School of Computer Science and Engineering	Date
Ernesto Gomez	
Keith Evan Schubert	

ABSTRACT

Image reconstruction of proton Computed Tomography (pCT) is a process of solving x within a linear equation $Ax = b$, where A is path matrix and b is electron density matrix. Iterative Reconstruction Techniques are widely used to generate relative electron density maps for proton therapy. The reordering subsets methods, which group projections data in a certain sequence, have been proposed. This project investigates the improvement of iterative reconstruction using reordered subsets.

The simulation result shows that reordering subsets image reconstruction algorithms achieve more accuracy image in the same or less number of reconstruction cycles. Further research on deploying this algorithm on multiprocessors would be suggested.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Haiyan Qiao for all her time, knowledge and patience. I would also like to thank my committee members, Dr. Keith Schubert and Dr. Ernesto Gomez, and my graduate coordinator Dr. Josephine Mendoza. I would also like to thank Dr. Richard Botting for his help. I would also like to thank my parents for their support and patience. I would also like to thank my Uncle Xinnong Zhou, his family, and my friend Xuelian.

DEDICATION

To My Parents

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

Generating accurate electron density maps in the shortest possible time is the goal of pCT. Due to the huge amount of data involved, direct methods on solving the large and sparse linear equation, $Ax = b$, is not feasible. Iterative reconstruction algorithms are used for pCT image reconstruction. Based on Simultaneous algebraic reconstruction techniques (SART), block-iterative SART is employed. This project explores the possibility of accelerating pCT image reconstruction by reordering the sparse matrix A.

1.1 Background

Proton therapy has a significant difference from conventional radiation treatment that the energy distribution of protons can be directly placed in tissue volumes of any desired depth. This increases the control of tumor while highly reducing unnecessary damage to surrounding healthy tissues. Currently, proton treatment doses are calculated using data from X-ray computed tomography (xCT), which cannot predict an accurate high dosage peak, which is the Bragg peak, for treatment. By measuring the energy loss of protons, proton computed tomography (pCT) provides more accurate proton doses calculations with lower radiation doses and verifies treatment position in patient relative to the correct proton beam. Meanwhile, pCT offers the possibility

for on-line treatment planning for proton therapy [6]. However, a fully operational pCT system does not exist currently [7] due to the large size of proton histories (100 million by 30 million for a human head and neck [7]) that are collected for object image reconstruction.

Proton CT has been explored in the last decade [6, 8, 10].

1.2 Purpose

Block iterative projection and Ordered Subset reconstruction algorithms are developed to improve the performance of image reconstruction. In the previous research [12, 8], updates of voxels are calculated every subset of proton histories. However, how to order the proton histories to group them into subsets has not been explained. In this case, it brought me the idea to investigate the possibility of improvement of pCT reconstruction by reordering the path matrix before iterative reconstruction process. What type of proton data should be grouped together to form subsets? In order to update as many voxels as possible in each subset, it is needed to group the proton data blocks which have the least overlap[1].

1.3 Significance

As introduced in previous section, pCT aims at efficient computation and provision of accurate electron density maps [8]. Reordering the path data offers the opportunity to achieve better electron density resolution in less iterations than the reconstruction achieved from unordered path data. It means that it will reduce the waiting time for

[1] Overlap occurs when proton pass through a same voxel.

the CT image which need to be done in a shorter time for treatment planning and pre-treatment patient position verification images.

1.4 Flow of Document

In this document, the following will be shown:

- Iterative reconstruction algorithms in Chapter 2.
- Reordering subset methods and simulation with reordered subsets in Chapter 3.
- Analysis on simulation results in Chapter 3.
- Future works and conclusion in Chapter 4.

2. RECONSTRUCTION ALGORITHMS

Proton CT image reconstruction is modeled by the equation $Ax = b$, where A is a known sparse matrix representing the traverse of proton projections, b is the integral relative electron density which converted from energy loss, x is the unknown image need to be reconstructed. Direct methods on solving $Ax = b$ are not feasible due to the huge amount of path data (the size of A is about 30 billion for reconstructing a 2 dimension image with a 1000 by 1000 resolution).

Different Algebraic algorithms are proposed on solving this equation. In most of the algebraic implementations, path matrix A has been simplified by containing 1's and 0's based on whether the proton traverse through that voxel or not.

2.1 Algebraic Reconstruction Technique (ART)

The first iterative algorithm is algebraic reconstruction technique (ART), which is a full sequencial method that updates image x with every proton history but due to inconsistencies [5], ART usually suffers from salt and pepper noise.

$$x^{k+1} = x^k + \lambda_k \frac{b_i - \left\langle a^i, x^k \right\rangle}{\|a^i\|^2} a^i \tag{2.1}$$

Where λ_k is a user-determined relaxation parameter sequence, which can be changed during the process of iterations.

2.2 Simultaneous Algebraic Reconstruction Technique (SART)

Developed on the base of ART, Simultaneous Algebraic Reconstruction Technique (SART) offers good quality reconstruction image and less numerical variation in one iteration [5]. This technique updates image x after going through all the proton histories. It improves the reconstruction accuracy while maintaining a rapid convergence.

$$x^{k+1} = x^k + \frac{\lambda_k}{m} \sum_{i=1}^{m} \frac{b_i - \left\langle a^i, x^k \right\rangle}{\|a^i\|_2^2} a^i \tag{2.2}$$

Where λ_k is a user-determined relaxation parameter sequence, m is equal to the number of proton histories.

2.3 Ordered Subset Simultaneous Algebraic Reconstruction Technique (OS-SART)

Ordered Subset Simultaneous Algebraic Reconstruction Technique groups proton histories from different projections and updates image x after going through all the proton histories in one subset. This technique reduces the noise in the reconstruction image. I use OS-SART as the reconstruction algorithm to test the result of reordering subsets, which will be shown in the following chapters.

$$x_j^{k+1} = x_j^k + \left(\frac{\lambda_k}{\sum_{i \in I_{t(k)}} a_j^i} \right) \sum_{i \in I_{t(k)}} \frac{b_i - \left\langle a^i, x^k \right\rangle}{\sum_{j=1}^{n} a_j^i} a_j^i \tag{2.3}$$

Where λ_k is a user-determined relaxation parameter sequence.

3. REORDERING SUBSET METHODS

In order to improve the quality of image or reduce the reconstruction time, updating as many voxels as possible through every subset in OS-SART is needed. Therefore, I try to reorder path matrix by grouping proton histories which have the least overlap into a new subset, where an overlap occurs when two proton traverse through the same voxel in the object. I proposed two methods on finding the least overlap between different blocks of path data.

3.1 Reordering path matrix

In the path matrix A, $a^i (i \in 1, 2, \ldots, m)$ represents the traverse route of ith proton. The inner product of two rows in path matrix is equal to the overlap of these two proton histories.

$$Overlap = \left\langle a^{i_1}, a^{i_2} \right\rangle \tag{3.1}$$

Since all protons are randomly projected in a same angle of projection, there is not a certain pattern of ordering for a^i in path matrix A. The solution of overlap of two projections should be:

$$Overlap = \sum_{i \in I_t(k)} \sum_{j \in I_t(k')} \left\langle a^i, (a^j)^T \right\rangle \tag{3.2}$$

where $I_t(k)$ is the index of rows in one projection.

By comparing all the projections, the two projections which have the minimum overlap are grouped into a new subset to update x. After one reordering one subset finished, grouping other subsets from the left projections in path matrix is continued. The number of projections gathered into a new subset can be determined by users. However, this method will be such a time consuming process that will significantly increase the total time of image reconstruction. In order to reduce the reordering time to a reasonable period, the following two methods are proposed.

3.1.1 Full Search Reordering (FSR)

The first method is inspired by block matching algorithm generally deployed in video compression [11]. Summation of the inner product of each two corresponding rows(for instance, same index in different projections) is computed only.

$$Full\ Search\ Overlap = \sum_{i \in I} \left\langle a^i_{ref}, a^i_{cur} \right\rangle \tag{3.3}$$

Where $reference(ref)$ and $current(cur)$ projections are the projections which are calculating the overlap. In advance, a simplified FSR that only search half of the left projections randomly can be tested in the future work which may offer a good enough reconstruction as well as reducing the reordering time into half of the time used for FSR.

Pseudocode of full search reordering

The pseudocode of FSR is listed as below:

{A is a M*N sparse path matrix

$a[i][j]$ is the element in A

s_r_1 is number of rays in 1 projection

s_r_m is number of rays in 1 micro block(set)

num_subset is how many ordered subset you want to create

um_p_re is how many projections you want to take to do the reordering

$total\_projections\}$

$fullsearchPathMat(A, num\_sebset, num\_p\_re, s\_r\_1, s\_r\_m)$

$M \leftarrow number\ of\ rows\ of A$

$N \leftarrow number\ of\ columns\ of A$

$\{i\_ref$ is row index in the reference projection$\}$

for $i\_ref \leftarrow 0$ to s_r_1 **do**

  $minimum \leftarrow M * N$

  $i\_min \leftarrow 0$

  $subset \leftarrow subset + micro\_block[i\_ref]$

  $\{i\_cur$ is row index in the current projection$\}$

  for $P \leftarrow 0$ to num_p_re **do**

    $i\_cur \leftarrow i\_ref + num\_p\_re * s\_r\_1$

    while $i\_cur < (num\_p\_re + 1) * s\_r\_1 - s\_r\_m$ **do**

      $sum \leftarrow \langle a[i\_ref], a[i\_cur] \rangle$

      if $sum < minimum$ **then**

        $minimum \leftarrow sum$

        $i\_min \leftarrow i\_cur$

      end if

end while{go thru all the micro blocks in current projection} {should have one maximum and corresponding i_min}

$subset \leftarrow subset + micro\_block[i\_min]$

$i \leftarrow i + s\_r\_m$

end for

end for

Return subset

3.1.2 Sum Search Reordering (SSR)

The second method is sum search reordering (SSR). Instead of compute inner product of all the element in the different projections, this algorithm use the vector of summation of each column in a project to represent this group of protons traverse area in the object. By comparing the difference of two sum vectors of two projections in path matrix, it can be told that the bigger difference these two vectors get, the less overlap these two projection have.

$$Sum\ Search\ Overlap = \sqrt{\sum_{j=0}^{N} \left(\sum_{i \in I(ref)} a_j^i - \sum_{i \in I(cur)} a_j^i \right)^2} \tag{3.4}$$

Where $reference(ref)$ and $current(cur)$ projections are the two projection which computing the overlap.

Pseudocode of sum search reordering

The pseudocode of SSR is listed as below:

{A is a M*N sparse path matrix

$a[i][j]$ is the element in A

s_r_1 is number of rays in 1 projection

s_r_m is number of rays in 1 micro block(set)

num_subset is how many ordered subset you want to create

um_p_re is how many projections you want to take to do the reordering

$total\_projections$

Discription: comparing the difference of sum of each columns between two micro blocks. Find the largest different means they have the least overlap between these two micro blocks. }

$sumsearchPathMat(A, num\_sebset, num\_p\_re, s\_r\_1, s\_r\_m)$

$M \leftarrow number\ of\ rows\ of A$

$N \leftarrow number\ of\ columns\ of A$

{i_ref is row index in the reference projection}

for $i\_ref \leftarrow 0$ to s_r_1 **do**

  $maximum \leftarrow 0$

  $i\_max \leftarrow 0$

  $subset \leftarrow subset + micro\_block[i\_ref]$

  $Sum\_ref[N] \leftarrow eachcolumnssumofrefblock$

  {i_cur is row index in the current projection}

  for $P \leftarrow 0$ to num_p_re **do**

    $i\_cur \leftarrow i\_ref + num\_p\_re * s\_r\_1$

    while $i\_cur < (num\_p\_re + 1) * s\_r\_1 - s\_r\_m$ **do**

      $var \leftarrow 0$

      $Sum\_cur[N] \leftarrow eachcolumnssumofcurblock$

$var \leftarrow \|Sum\_ref[N] - Sum\_cur[N]\|$

if $var > maximum$ **then**

$maximum \leftarrow var$

$i\_max \leftarrow i\_cur$

end if

end while{go thru all the micro blocks in current projection} {should have one maximum and corresponding i_max}

$subset \leftarrow subset + micro\_block[i\_max]$

$i \leftarrow i + s\_r\_m$

end for

end for

Return subset

3.2 Simulation

Currently in pCT system, path matrix A is generated by Most Likely Path (MLP) [7] algorithm. MLP converts path data, which contains the pre- and post-object coordinates into a sparse matrix A. Every row in A is the vectorized traverse route of each proton. b is the integral relative electron density which converted from energy loss measured during the proton projection. In this project, path matrix A is simulated by using a random sparse matrix which contains only 0's and 1's, since protons are randomly projected in each projection from a same angle. In this experiment, all recorded time data are the average time of running the process 5 times. The simulation process is listed as following:

1. Generate a random sparse matrix A.

2. Use a known picture as $actualx$, then get $b = A * actual\ x$.

3. Reorder A and b by using reordering algorithms, get $new\ A$ and $new\ b$ as output.

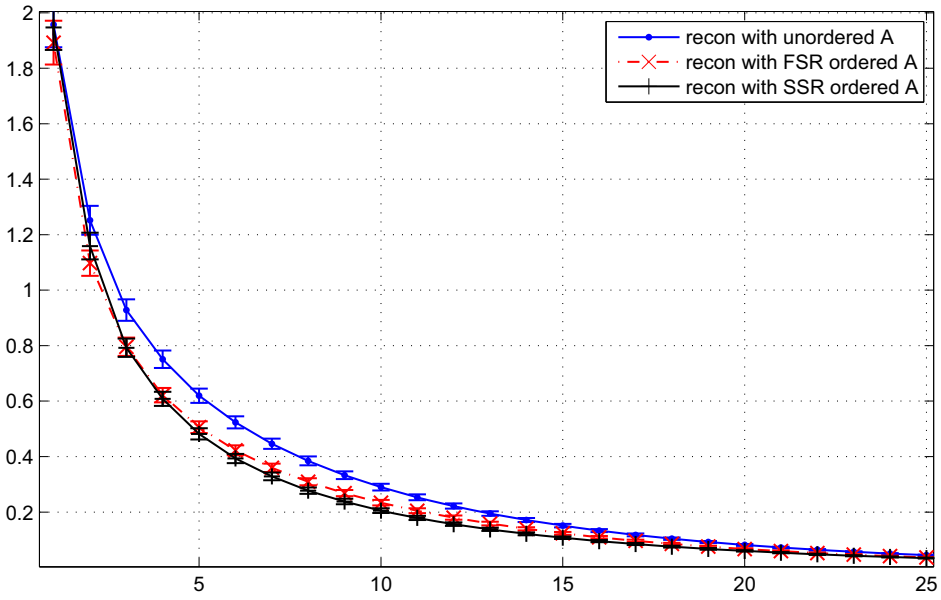
4. Reconstruct the first image x with OS-SART by using $new\ A$ and $new\ b$. Reconstruct the second image x with OS-SART using A and b.

5. Analyse the result of simulation.

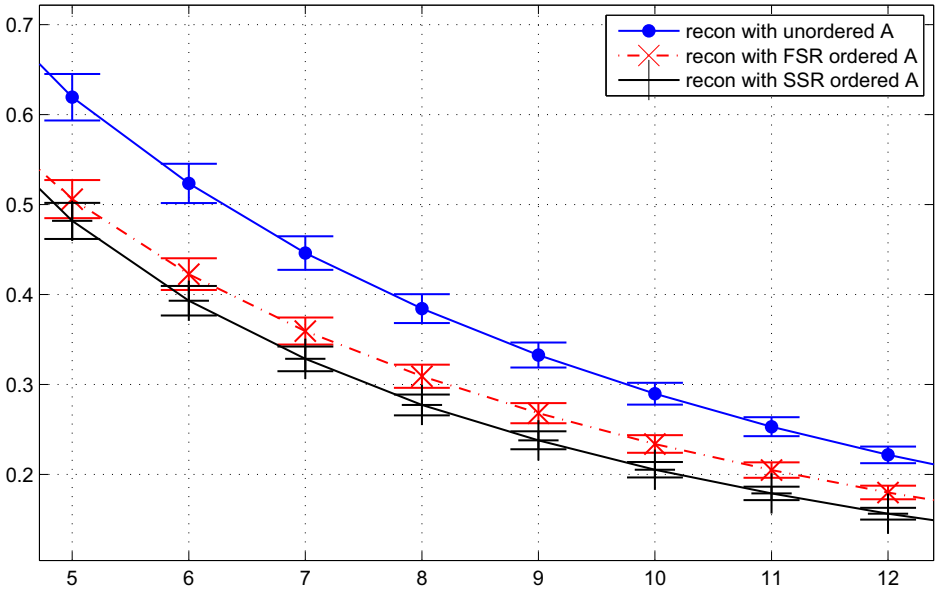
3.3 Analysis

By comparing the relative error of reconstructions in Figure 3.1, reordering OS-SART achieves better image with less error in less iterations. Based on Figure 3.1(a),it can be seen that, in iteration 5, OS-SART with reordered path matrix A offers better relative error than the error reconstructed with unordered A in iteration 6. It may save more than 5 minutes in real clinical situation which has much larger proton data than the data in this simulation. From Figure 3.1(b), it can be seen that the error of SSR offers even less error than FSR does in the same iteration number. These results are also summarized in Table 3.1 which includes iteration numbers, the relative error, and improvement percentage of both FSR and SSR. SSR OS-SART yields a 29.15% error improvement percentage when reconstruting 10 iterations while FSR OS-SART yields 19.27%. And these improvements will keep increasing with even more iterations.

In Figure 3.2, it can be seen that, in different size of path matrix A, FSR reordering time increases significantly while SSR reordering time keeps much less than the time



(a)



(b)

Fig. 3.1: Error of iteration number for OS-SART using unordered A, FSR ordered A, and SSR ordered A

Tab. 3.1: Relative error of iteration number for OS-SART using unordered A, FSR ordered A, and SSR ordered A

Iter	Unordered A(std)	FSR(std)	SSR(std)	FSR Improv	SSR Improv
2	1.252(.0522)	1.097(.0458)	1.159(.0483)	7.42%	12.33%
3	.9284(.0387)	.7960(.0332)	.7922(.033)	14.27%	14.68%
4	.7508(.0313)	.6215(.0259)	.6083(.0254)	17.22%	18.98%
5	.6193(.0258)	.5061(.0211)	.4819(.0201)	18.29%	22.19%
6	.5235(.0218)	.4227(.0176)	.3931(.0164)	19.25%	24.9%
7	.4461(.0186)	.3594(.0150)	.3285(.0137)	19.43%	26.36%
8	.3843(.0160)	.3091(.0129)	.2772(.0116)	19.56%	27.87%
9	.3328(.0139)	.2681(.0112)	.2380(.0099)	19.42%	28.48%
10	.2897(.0121)	.2339(.0098)	.2052(.0086)	19.27%	29.15%

of OS-SART. The data is listed in Table 3.2 also. Based on Table 3.2, SSR time stays less than 1 iteration time of OS-SART. With on the previous data, it is able to infer that with the same total reconstruction time, SSR OS-SART will achieve a better quality image, or it can be said that SSR OS-SART will need less time to achieve the same image quality than the time of OS-SART reconstructing from unordered path matrix A.

Figure 3.3 and Figure 3.4 display the reconstruction image of unordered, FSR, and SSR OS-SART with different iteration number.

Figure 3.5 and Figure 3.6 display the reconstruction image of unordered, FSR, and SSR OS-SART with different iteration number. This circle shape object also test the

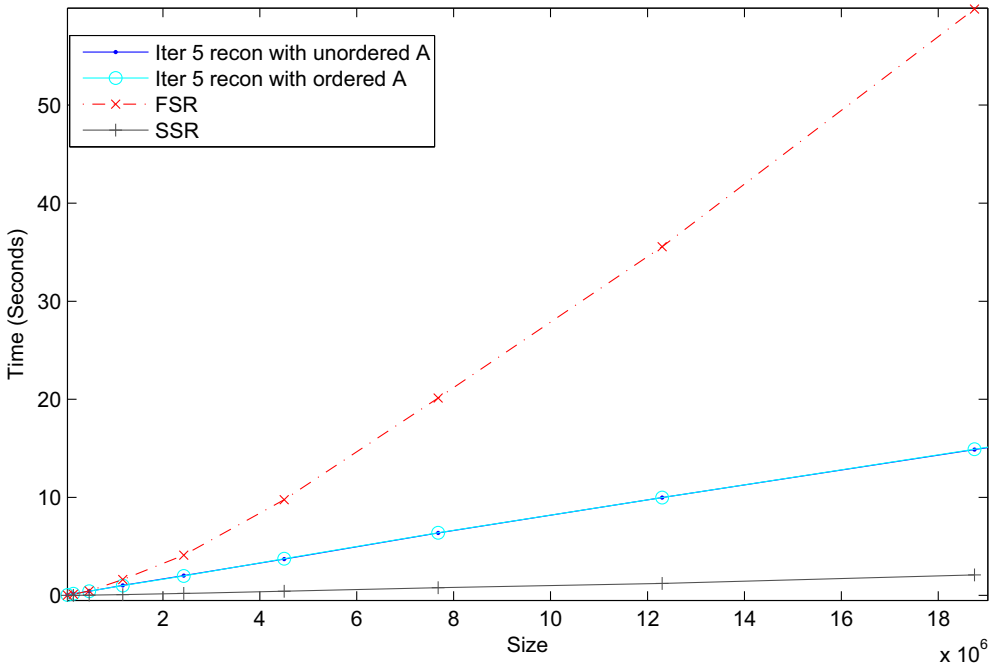
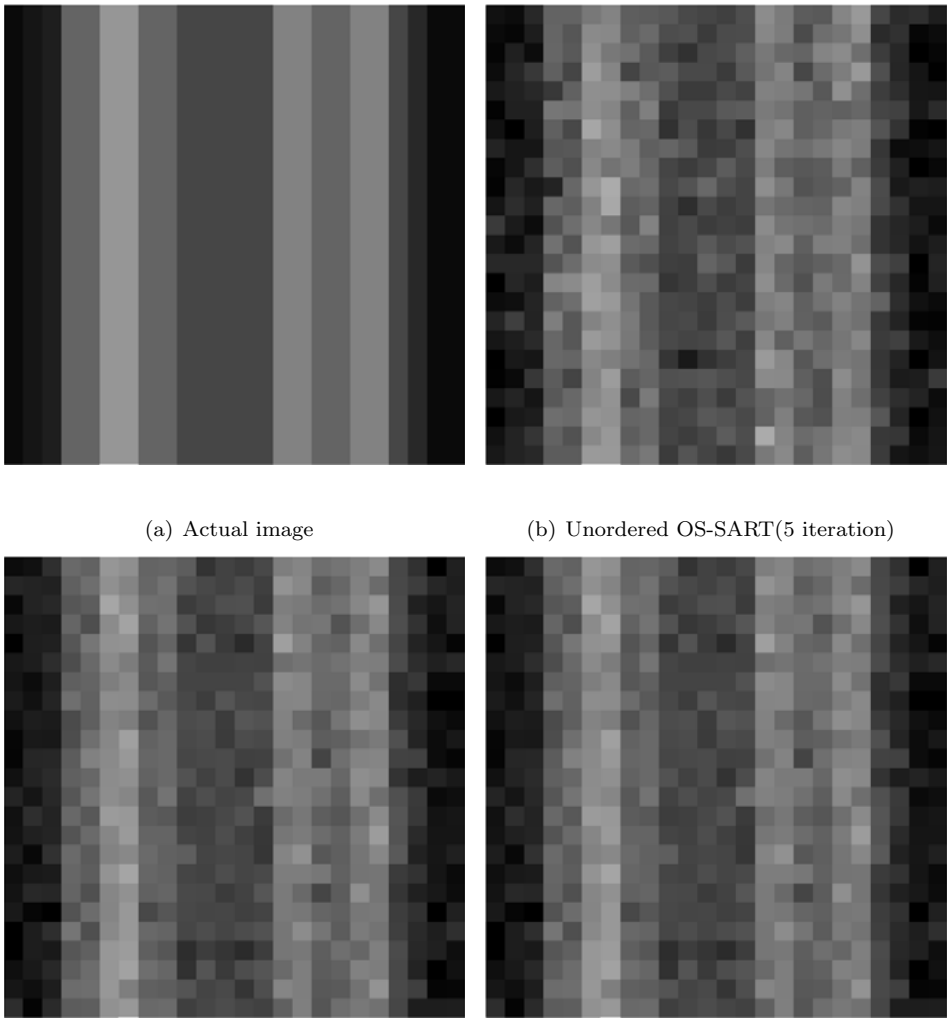


Fig. 3.2: Time of reordering methods and 5 iteration OS-SART

Tab. 3.2: Time of 1 iteration reordering methods and 1 iteration OS-SART

Size(10^6)	SSR time(sec)	OS-SART time(sec)
2.43	0.2028	0.3964
4.5	0.4304	0.7472
7.68	0.7925	1.2768
12.3	1.229	1.9948
18.75	2.099	2.982



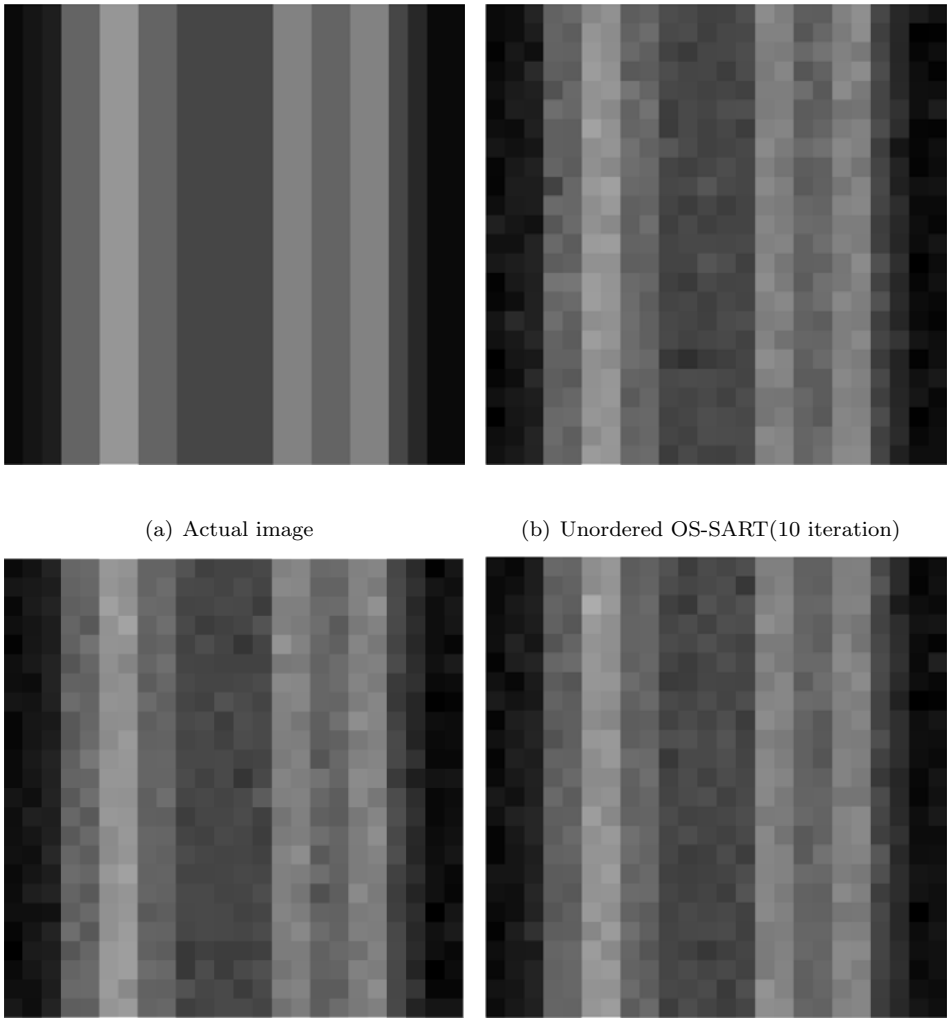
(a) Actual image

(b) Unordered OS-SART(5 iteration)

(c) FSR OS-SART(5 iterations)

(d) SSR OS-SART(5 iterations)

Fig. 3.3: Reconstructed bar images with unordered A, FSR ordered A, and SSR ordered A.



(a) Actual image

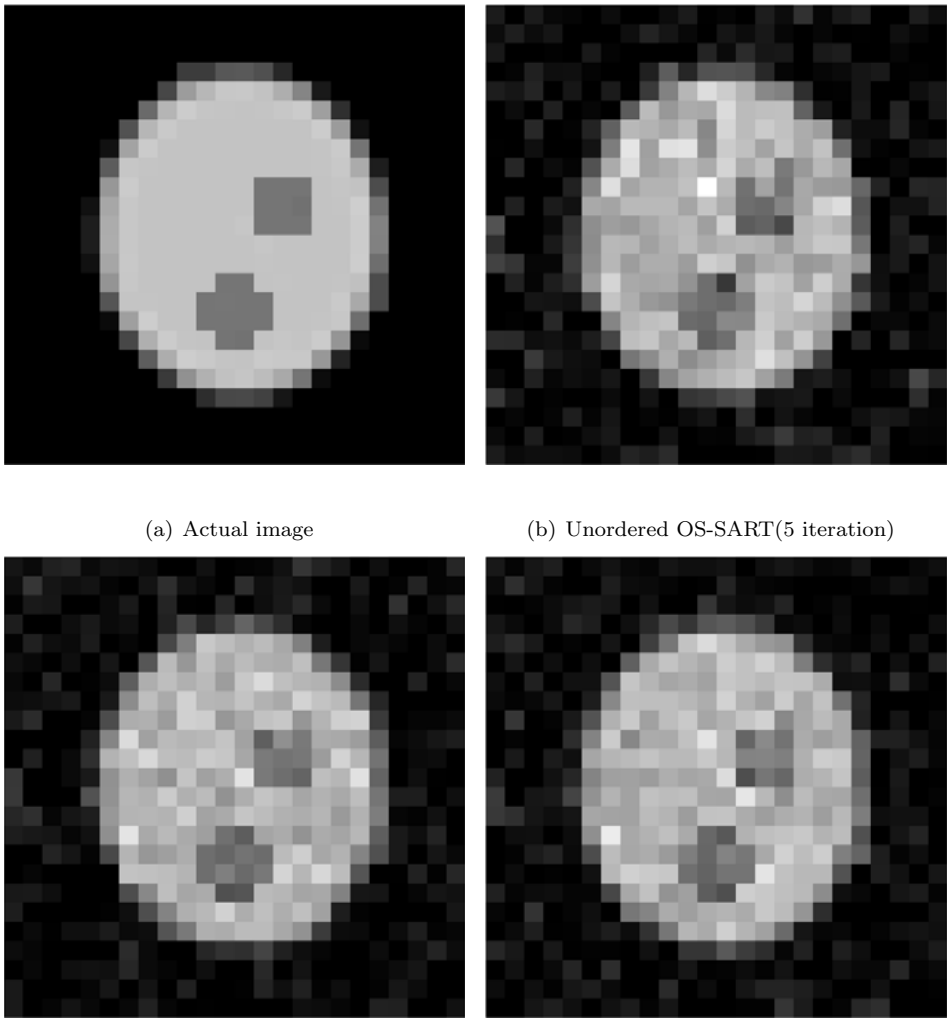
(b) Unordered OS-SART(10 iteration)

(c) FSR OS-SART(9 iterations)

(d) SSR OS-SART(8 iterations)

Fig. 3.4: Reconstructed bar images with unordered A, FSR ordered A, and SSR ordered A.

performance of reordering OS-SART on reconstructing the edge of objects.



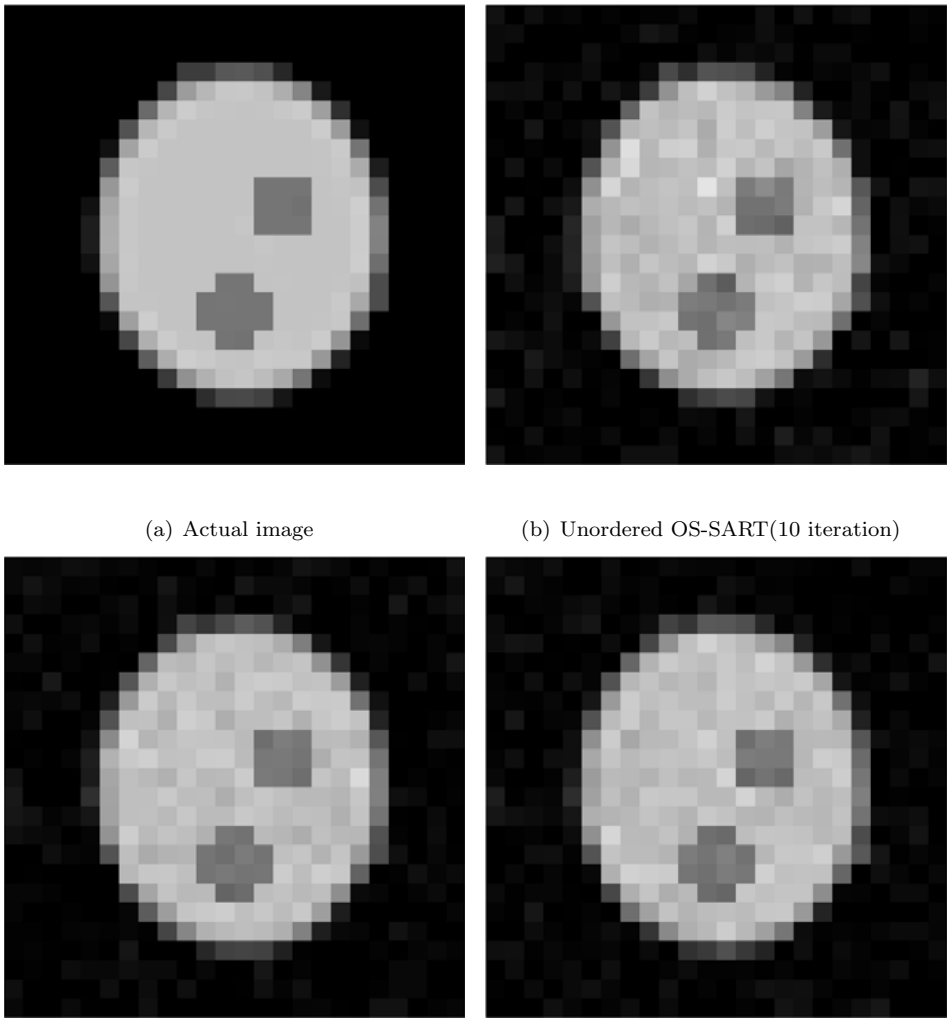
(a) Actual image

(b) Unordered OS-SART(5 iteration)

(c) FSR OS-SART(5 iterations)

(d) SSR OS-SART(5 iterations)

Fig. 3.5: Reconstructed circle images with unordered A, FSR ordered A, and SSR ordered A.



(a) Actual image

(b) Unordered OS-SART(10 iteration)

(c) FSR OS-SART(10 iterations)

(d) SSR OS-SART(10 iterations)

Fig. 3.6: Reconstructed circle images with unordered A, FSR ordered A, and SSR ordered A.

4. SUMMARY

4.1 Future Works

The following items show works

- Improve FSR and SSR algorithms to reduce the time of reordering path matrix A.
- Deploy SSR reordering algorithms to other block-iterative reconstruction techniques.
- Use multi-processors to accelerate the whole process of reordering subset image reconstruction.

4.2 Conclusion

In this project, experiments are simulated using different size, densities, and shapes of objects. Reordering subset image reconstruction algorithm achieved more accuracy image in the same or less reconstruction cycles. The reordered subsets can be assigned to multi-processor More efficient reordering methods may be developed by improving the two existing reordering algorithms in future.

APPENDIX A

SPARSE MATRIX COMPRESSION

Usually, sparse matrix is stored in a space saving format which reduce the storage space of sparse matrix significantly. All these formats only contain the non-zero elements of sparse matrix. The following are two mostly used formats.

A.1 Compressed Row Storage (CRS)

Sparse matrix is represented with three vectors. The first vector A contains the value of non-zero element. The second vector AJ contains the column position of corresponding non-zero elements in vector A. Each element in vector AI points to the first non-zero element of each row in vector A and AJ. For instance, a sparse matrix

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 1 \\ 0 & 3 & 7 & 0 & 0 \\ 0 & 0 & 4 & 1 & 0 \\ 5 & 0 & 0 & 2 & 0 \end{pmatrix}$$

is represented by three vectors shown bellow.

$A = [1\ 2\ 1\ 3\ 7\ 4\ 1\ 5\ 2]$,

$AJ = [0\ 1\ 4\ 1\ 2\ 2\ 3\ 0\ 3]$,

$AI = [0\ 3\ 5\ 7]$.

A.2 Jagged Diagonal (JD)

Jagged Diagonal format reorders the non-zero elements in sparse matrix by column. A sparse matrix will be represented in the following. Vector A contains the non-zero elements of sparse matrix as well as vector AJ contains the corresponding column index of A. Since the rows are reordered, vector I contains the reordered index of

rows.

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 \\ 0 & 3 & 7 & 0 & 1 \\ 0 & 0 & 4 & 0 & 0 \\ 5 & 0 & 0 & 2 & 0 \end{pmatrix}$$

is stored as

$$A = \begin{pmatrix} 3 & 7 & 1 \\ 1 & 2 & \\ 5 & 2 & \\ 4 & & \end{pmatrix}$$

.

$$AJ = \begin{pmatrix} 1 & 2 & 4 \\ 0 & 1 & \\ 0 & 3 & \\ 2 & & \end{pmatrix}$$

.

$$I = \begin{pmatrix} 1 \\ 0 \\ 3 \\ 2 \end{pmatrix}$$

.

APPENDIX B

BLOCK MATCHING ALGORITHMS IN VIDEO COMPRESSION

Block matching is the most popular method for motion estimation [11]. This method searches the most matching block of the same size in two different frames by calculating the mean absolute difference (MAD) of corresponding pixels. By considering one block in reference frame, a full search will search all the same size blocks in another frame. In order to optimize the search algorithm, several algorithms are proposed, such as three step search and cross search.

B.3 Three Step Search

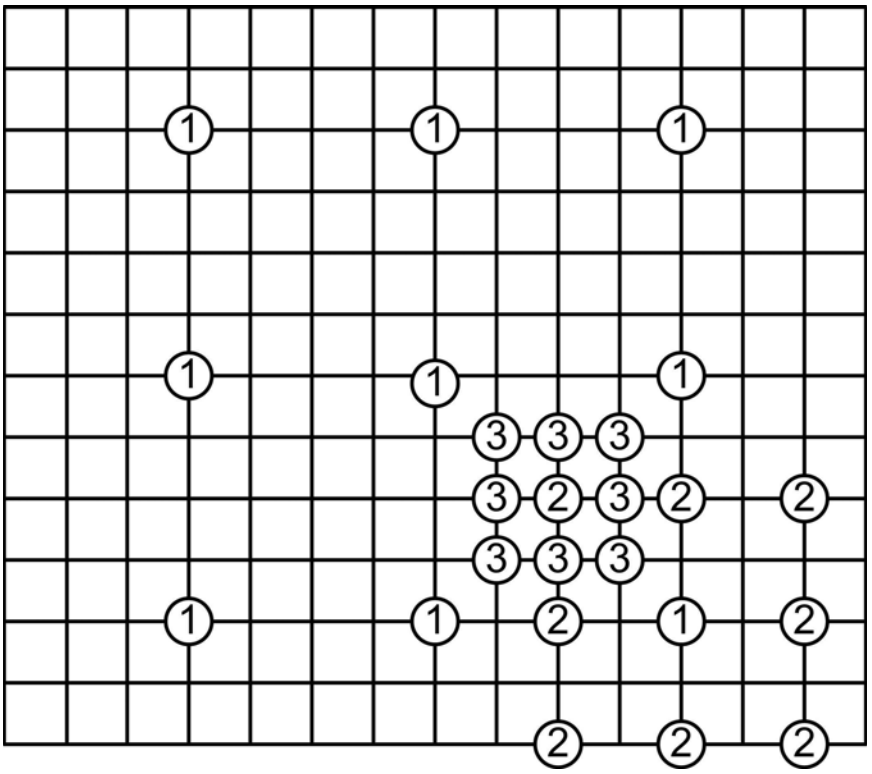
The procedure of three step search is listed as following.

1. MAD is evaluated at 9 blocks, which are marked as 1 in Figure B.3, around center block. A minimum MAD block will be found in this step.

2. 8 blocks, which are marked as 2, are evaluated which are around the minimum block found in the first step. A minimum MAD block will be found in this step.

3. In this step, the minimum MAD block will be found by evaluating the next 8 blocks, which are marked as 3, just next to the previous minimum MAD block.

The Figure B.3 also predicts the procedure of three step searching.


1 1 1
1 1 1
3 3 3
3 2 3 2 2
3 3 3
1 1 2 1 2
2 2 2

APPENDIX C

SOURCE CODE

This contains the source code used to pre-order the path matrix as well as reconstruct image by SART and OS-SART.

The following is the code for generating random sparse matrix.

```
N = 24 * 24;
M = N * 3;
fprintf ( 1, 'Going to generate a %4d * %4d Matrix.\n', M, N);
i = [];
j = [];
v = [];
m = M;
n = N;

a = sparse ( i, j, v, m, n );

fprintf ( 1, '\n' );
fprintf ( 1, '     I     J  New A(I,J)\n' );
fprintf ( 1, '\n' );
nonzeros = m * n * 0.02;
fprintf(1, 'number of nonzero: %4d %4d', nonzeros, floor(nonzeros));
for test = 1 : round(nonzeros)

  i = round ( m * rand ( ) + 0.5 );
  j = round ( n * rand ( ) + 0.5 );
  a(i,j) = 1;
  % fprintf ( 1, '  %4d  %4d  %f\n', i, j, a(i,j) );

end

fprintf ( 1, '\n' );
fprintf ( 1, '  Number of nonzero entries is %d\n', nnz ( a ) );
```

```
  A = full ( a );


  save('test_A_02.mat', 'A')
  spy(A);
  v = sum(A);
  min(v)
  %v
%   type test_A.mat
```

The following is the code for reordering path matrix with Full Search Reordering (FSR).

```
%% reordering matrix A and b by block matching among projections
% Discription: A full search for block matching
% Input: test_A_0*.mat -> A, test_b_0*.mat -> b
% Output: new_A -> test_new_A_0*.mat, new_b -> test_new_b_0*.mat
% Author: Wenzhe Xue
%
function BM_full_search()
    %load test matrix
    load('test01/test_A_01.mat') % A
    load('test01/test_b_01.mat') % b


    nz = nnz(A);
    [rows cols] = size(A);
    fprintf('rows = %d, columns = %d \n', rows, cols);
    fprintf('nonzeros of A = %d\n', nz);
    %s_r_1 = size of rays in 1 projection
    s_r_1 = 72;
    %spy(b), title('test path matrix A')
    %xlabel(sprintf('nonzeros = %d' , nz));
```

```
%setup a flag vector for searching projections
num_proj = 24;
for temp_i = 1:num_proj
    v_proj(1:temp_i) = 1;
end%end for
%v_proj
fprintf('1) Initial all the unsearched projection to 1.\n');
fprintf('   later if a projection is selected and copied to the new_A\n');
fprintf('   that proj will set to 0.\n');

%new_A_p is the pointer for inserting projs into new_A
new_A_p = 2;

%go thru half of the projection for ref projection
tStart = tic;
minustime = 0;
for ref_p = 1:num_proj
    fprintf('new_A_p = %d \n', new_A_p);

    if v_proj(ref_p)== 1

%%*********************************************************************
%get the reference projection
ref = A((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
%flag 0 to ref proj
v_proj(ref_p) = 0;
%
minimum = s_r_1 * cols;
fprintf('===Initial minimum = %d for ref_p[%d]===\n', minimum, ref_p);
for p = ref_p+1 : num_proj %p go thru cur proj
    if v_proj(p) == 1
        %%*************************
        cur = A(((p-1) * s_r_1+1) : (p * s_r_1),:);
```

```
        % finding the overlap between ref projection and
        % current projectoin
        for i = 1:s_r_1
            for j = 1:cols
              s(i,j) =ref(i,j) && cur(i,j);
            end%end for
        end%end for
        % nnz in the S matrix is the number of overlap
        % after && operation
        sum = nnz(s);

        if( sum ≤ minimum )
            minimum = sum;
            min_block = p;

            min_block_i = p*s_r_1+1;
        end%end if
        %%**************************
    else
        fprintf('Nothing: Reached a projection already selected.\t');
    end%end if
end%end for
% print out the cur_i and number of overlap after a full search
fprintf('\nOne full search finished.\ncur_i = %d, minimum = %d\n',
          min_block_i, minimum);

%flag 0 to projection p
v_proj(min_block) = 0;
fprintf('Flag projection %d to 0\n', min_block);

% after find the min_proj, put these part of matrix A into the new
% matrix
if nnz(v_proj)==2
```

```
    fprintf('Only 2 projs left.\n');
end%end if

tStorage = tic;
%reorder two projections to new A
new_A((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) = ref;
new_A(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                            A(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);
fprintf('Pushed %d and %d projs into new_A\n',ref_p, min_block);

%change the b_i of this proton history to the corresponding place
new_b((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) =
                            b((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
new_b(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                            b(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);
%everytime u push projections, new_A_p++
new_A_p = new_A_p + 2;
%fprintf('new_A_p = %d after increase.\n', new_A_p);
%v_proj
tS = toc(tStorage);
minustime= minustime+tS;
%spy(new_A);
%%*****************************************************************
    end%end if
end%end for

totaltime = toc(tStart);
fprintf('finish all the search in %4d, v_proj should set to 0\n', totaltime);
%v_proj
nz_new = nnz(new_A);
if nz ≠ nz_new
    fprintf('nonzeros of A     = %d\n', nz);
    fprintf('nonzeros of new A = %d\n', nz_new);
    fprintf('DANG, seems not correct¬!\n');
```

```
    else
        disp('Finish all. Oh Yeah¬');
        save('test_new_A_01.mat', 'new_A');
        save('test_new_b_01.mat', 'new_b');
    end%end if
end
```

The following is the code for reordering path matrix with Sum Search Reordering (SSR).

```
%% reordering matrix A and b by block matching among projections
% Discription: find the difference of sum of each columns between
%              two blocks. the largest difference means they have
%              the least overlap
% Input: test_A_0*.mat -> A, test_b_0*.mat -> b
% Output: new_A -> test_new_A_0*.mat, new_b -> test_new_b_0*.mat
% Author: Wenzhe Xue
%
function sum_search()
% for l = 1: 100
    %load test matrix
    load('test_A_01.mat') % A
    load('test_b_01.mat') % b

    nz = nnz(A);
    [rows cols] = size(A);
    fprintf('rows = %d, columns = %d \n', rows, cols);
    fprintf('nonzeros of A = %d\n', nz);
    %s_r_1 = size of rays in 1 projection
    s_r_1 = 72;
    %spy(b), title('test path matrix A')
```

```
%xlabel(sprintf('nonzeros = %d' , nz));



%setup a flag vector for searching projections
num_proj = 24;
for temp_i = 1:num_proj
    v_proj(1:temp_i) = 1;
end%end for
%v_proj
fprintf('1) Initial all the unsearched projection to 1.\n');
fprintf('   later if a projection is selected and copied to the new_A\n');
fprintf('   that proj will set to 0.\n');

%new_A_p is the pointer for inserting projs into new_A
new_A_p = 1;

%watch the time on storage the new A+++++++++++++++++
minustime= 0;
%go thru half of the projection for ref projection
tStart = tic;
for ref_p = 1:num_proj
    fprintf('new_A_p = %d \n', new_A_p);

    if v_proj(ref_p)== 1

%*********************************************************************
%get the reference projection
ref = A((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
%sum ref projection
sum_ref = sum(ref);
%flag 0 to ref proj
v_proj(ref_p) = 0;
%
maximum = 0;
```

```
fprintf('===Initial maximum = %d for ref_p[%d]===\n', maximum, ref_p);
for p = ref_p+1 : num_proj %p go thru cur proj
    if v_proj(p) == 1
        %%**************************
        cur = A(((p-1) * s_r_1+1) : (p * s_r_1),:);
        sum_cur = sum(cur);

        % finding the overlap between ref projection and
        % current projectoin
        norm1 =norm(sum_ref - sum_cur);
        if( norm1 ≥ maximum )
            maximum = norm1;
            min_block = p;

            min_block_i = p*s_r_1+1;
        end%end if
        %%**************************
    else
        fprintf('Nothing: Reached a projection already selected.\t');
    end%end if
end%end for
% print out the cur_i and number of overlap after a full search
fprintf('\nOne full search finished.\ncur_i = %d, maximum = %d\n',
        min_block_i, maximum);

%flag 0 to projection p
v_proj(min_block) = 0;
fprintf('Flag projection %d to 0\n', min_block);

% after find the min_proj, put these part of matrix A into the new
% matrix
if nnz(v_proj)==2
    fprintf('Only 2 projs left.\n');
end%end if
```

```
tStorage = tic;
%reorder two projections to new A
new_A((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) = ref;
new_A(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                            A(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);
fprintf('Pushed %d and %d projs into new_A\n',ref_p, min_block);

%change the b_i of this proton history to the corresponding place
new_b((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) =
                            b((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
new_b(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                            b(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);
%everytime u push projections, new_A_p++
new_A_p = new_A_p + 2;
%fprintf('new_A_p = %d after increase.\n', new_A_p);
%v_proj
tS = toc(tStorage);
minustime= minustime+tS;
%spy(new_A);
%*****************************************************************
    end%end if
end%end for

totaltime = toc(tStart);
totaltime = totaltime - minustime;
fprintf('time for storage = %f seconds.\n', minustime);
fprintf('finish all the search in %f seconds.\n', totaltime);
%v_proj
nz_new = nnz(new_A);
if nz ≠ nz_new
    fprintf('nonzeros of A     = %d\n', nz);
    fprintf('nonzeros of new A = %d\n', nz_new);
    fprintf('DANG, seems not correct¬!\n');
else
```

```
        disp('Finish all. Oh Yeah¬');
%         save('sum_new_A_01.mat', 'new_A');
%         save('sum_new_b_01.mat', 'new_b');
    end%end if
end
% plot(totaltime);
%
% end %end forloop
```

The following is the code of OS-SART.

```
% SART
% author: Wenzhe Xue

function OSSART()

% load A and b
load('test_A_01.mat'); % new_A
%load('test_b_011.mat'); % new_b

%load x, for testing only
%load('test_x_smooth.mat');
load('circle_x.mat');
b = A*x;
   %x_true = reshape(x_reshape, 6, 6);
   %x_true(6,:)
   %b = A * x_reshape;
   %b

[rows cols] = size(A);

% x is the vector for contain the update for each iteration
% initial x_0 = 0
```

```
x_iter = zeros(cols, 1);
% var1 is the vector contains b_i - <A_i, x_k>/||A_i||_2^2
% 2-norm of A_i = sqrt(nnz(A_i)), so ||A_i||_2^2 = nnz(A_i)
var1 = zeros(rows, 1);
var2 = zeros(1, cols);

iter = 10; error = zeros(1, iter);
weight = 1.5;

fprintf('Iteration begin, will run total %d iterations. Weight = %d. \n',iter, weight);
tic;
t = 144;
for k = 1 : iter
    for os = 1: ceil(rows/t)
        for i = (os-1)*t+1:os*t
            if i < rows
                var1(i)=(b(i) - A(i,:)*x_iter) / nnz(A(i, :));
                for j = 1: cols
                    if A(i,j) ~= 0
                        var2(1,j)= var2(1,j)+var1(i)*A(i,j);
                    end
                end
            end
        end
        if i > rows
            var_div = sum(A((os-1)*t+1:rows,:));
        else
            var_div = sum(A((os-1)*t+1:os*t,:));
        end
        for j = 1: cols
            if var_div(j) ~= 0
                var2(1,j) = var2(1,j)/var_div(1,j);
            end
        end
```

```
        % after the above for loop, var2 = sum((b_i - <A_i, x_k>/||A_i||_2^2)*A_i)
        x_iter = x_iter + weight/os * var2';
        fprintf('%d%%...\n',floor(k/iter*100));
        %for check the error
        %     diff = x_iter' - x_reshape;
        %     error(k) = norm(diff);

    end
    %finish 1 iteration
    error = x_iter - x;
    err(k) = norm(error)/cols;
end%end iteration
% plot(error);
time = toc;
%save('error.mat', 'error');
%fprintf('Time = %d.\n', t);
% size(x_reshape)
% size(x_iter')
x_iter = x_iter';
%axis([0 cols -20 20])
%plot(x_iter, '*');
%load('test_x.mat');
% load('test_x_smooth.mat');
plot(x_iter, 'DisplayName', 'x_iter', 'YDataSource', 'x_iter');
hold all;
plot(x', 'DisplayName', 'x_reshape', 'YDataSource', 'x_reshape');
%plot(sumA, '-o');
xlabel(sprintf('OS-SART Iter = %d, Time = %.6f sec \n Unordered',iter, t));
hold off;
%
%
% figure
imshow(reshape(uint8(x_iter),24,24));
%imshow(reshape(uint8(x),24,24));
```

```
disp('finish');
%plot(x_reshape, 'o');

end
```

The following is the code of SART.

```
% SART
% author: Wenzhe Xue

function SART()

% load A and b
load('test_A_01.mat'); % A
load('test_b_01.mat'); % b

[rows cols] = size(A);

% x is the vector for contain the update for each iteration
% initial x_0 = 0
x_iter = zeros(cols, 1);
% var1 is the vector contains b_i - <A_i, x_k>/||A_i||_2^2
% 2-norm of A_i = sqrt(nnz(A_i)), so ||A_i||_2^2 = nnz(A_i)
var1 = zeros(rows, 1);
var2 = zeros(1, cols);

iter = 50;
weight = 1.5;

fprintf('Iteration begin, will run total %d iterations. Weight = %d. \n',iter, weight);
tic;
for k = 1 : iter
    for i = 1:rows
```

```
        var1(i)=(b(i) - A(i,:)*x_iter) / nnz(A(i, :));
        for j = 1: cols
            var2(1,j)= var2(1,j)+var1(i)*A(i,j);
        end
    end

    % after the above for loop, var2 = sum((b_i - <A_i, x_k>/||A_i||_2^2)*A_i)
    x_iter = x_iter + weight/rows * var2';

    %x_iter'

    %finish 1 iteration
end%end iteration
t = toc;
%fprintf('Time = %d.\n', t);
% size(x_reshape)
% size(x_iter')
x_iter = x_iter';
%axis([0 cols -20 20])
%plot(x_iter, '*');
% load('test_x.mat');
% plot(x_iter, 'DisplayName', 'x_iter', 'YDataSource', 'x_iter');
% hold all;
% plot(x, 'DisplayName', 'x_reshape', 'YDataSource', 'x_reshape');
% %plot(sumA, '-o');
% xlabel(sprintf('Iter = %d, Time = %.6f sec',iter, t));
% hold off;
%
%
% figure(gcf)

disp('finish\n')
%plot(x_reshape, 'o');
```

```
% xx = reshape(x, 24, 24);
%
% load('test_x.mat');
%
% xxx = xx-x;

end
```

The following is the code for testing the performance of SSR and FSR on different size.

```
%function [ output_args ] = TimeSize( input_args )
%TIMESIZE Summary of this function goes here
%   Detailed explanation goes here
tsize = 11;
t1 = zeros(tsize,1);
t2 = zeros(tsize,1);
t3 = zeros(tsize,1);
index = 1;
for test_size = [10 : 5 : 60]
fprintf('round %d.\n', index);

%each size loop 5 times then compute the time / 5 to get average time.
for ave = 1: 5

test(index) = test_size;
%%
%--------------------------
% generate random sparse matrix A
  N = test_size * test_size;
  M = N * 3;
```

```
  %fprintf ( 'Going to generate a %4d * %4d Matrix.\n', M, N);
  i = [];
  j = [];
  v = [];
  m = M;
  n = N;

  a = sparse ( i, j, v, m, n );

  nonzeros = m * test_size;
  %fprintf(1, 'number of nonzero: %4d %4d', nonzeros, floor(nonzeros));
  for test = 1 : round(nonzeros)

    i = round ( m * rand ( ) + 0.5 );
    j = round ( n * rand ( ) + 0.5 );
    a(i,j) = 1;
    % fprintf ( 1, '  %4d  %4d  %f\n', i, j, a(i,j) );

  end

%   fprintf ( 1, '\n' );
%   fprintf ( 1, '  Number of nonzero entries is %d\n', nnz ( a ) );

  A = full ( a );
  clear a;
  new_A = zeros(m,n);
  %spy(a);
%--------------------------
% generate random matrix x
  x = (5*rand(n, 1)+5* rand(n,1))/2;
  b = A * x;
%%
%+++++++++++++++++++++++++++
% reordering matrix A & b
```

```
%  rows = m; cols = n;
   [rows cols] = size(A);
  %s_r_1 = size of rays in 1 projection
  s_r_1 = test_size * 3;
  %setup a flag vector for searching projections
  num_proj = rows/s_r_1;
  for temp_i = 1:num_proj
    v_proj(1:temp_i) = 1;
  end%end for

  %new_A_p is the pointer for inserting projs into new_A
  new_A_p = 1;

  %go thru half of the projection for ref projection
  tStart = tic;
  minustime = 0;
  for ref_p = 1:num_proj
    if v_proj(ref_p)== 1
      %get the reference projection
      ref = A((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
      %flag 0 to ref proj
      v_proj(ref_p) = 0;
      %fprintf('%d ', ref_p);
      %
      minimum = s_r_1 * cols;
      %min_block =;
      for p = ref_p+1 : num_proj %p go thru cur proj
        if v_proj(p) == 1
          cur = A(((p-1) * s_r_1+1) : (p * s_r_1),:);
          % finding the overlap between ref projection and
          % current projectoin
          for i = 1:s_r_1
            for j = 1:cols
              s(i,j) =ref(i,j) && cur(i,j);
```

```
            end%end for
          end%end for
          % nnz in the S matrix is the number of overlap
          % after && operation
          sumnz = nnz(s);
          if( sumnz ≤ minimum )
            minimum = sumnz;
            min_block = p;
            min_block_i = p*s_r_1+1;
          end %end if
%       else
%         fprintf('Nothing: Reached a projection already selected.\t');
        end % end if
      end %end for
      % One full search finished
      %flag 0 to projection p
      v_proj(min_block) = 0;
      %fprintf('%d ', min_block);
      tStorage = tic;
      %reorder two projections to new A
      new_A((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) = ref;
      new_A(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                              A(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);

      %change the b_i of this proton history to the corresponding place
      new_b((new_A_p-1)*s_r_1+1 : new_A_p*s_r_1,:) =
                              b((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
      new_b(new_A_p*s_r_1+1:(new_A_p+1)*s_r_1, : ) =
                              b(( (min_block-1) * s_r_1+1) : (min_block * s_r_1),:);

      new_A_p = new_A_p + 2;
      tS = toc(tStorage);
      minustime= minustime+tS;
    end% end if
```

```
  end% end for
  tReorder(index) = toc(tStart) - minustime;
  t1(index) = t1(index)+ tReorder(index);
  %fprintf('reordering time = %f seconds.\n', tReorder(index));
%   if(nnz(A) ≠ nnz(new_A))
%       fprintf('nah...\n');
%   end

%%
%+++++++++++++++++++++++++++
% reconstruct x with A & new_A
  % x is the vector for contain the update for each iteration
  % initial x_0 = 0
  x_iter = zeros(cols, 1);
  % var1 is the vector contains b_i - <A_i, x_k>/||A_i||_2^2
  % 2-norm of A_i = sqrt(nnz(A_i)), so ||A_i||_2^2 = nnz(A_i)
  var1 = zeros(rows, 1);
  var2 = zeros(1, cols);
  var_div = zeros(1, cols);
  %iter is how many iterations we are going to take
  iter = 10;
  error = zeros(1, iter);
  %error_w with ordering
  error_w = zeros(1, iter);
  weight = 1.5;

  t=s_r_1 * 2;

  %treconw time on OS-SART w/ reordering
  treconw = tic;
  %OS-SART w/ ordering
  for k = 1 : iter
    for os = 1: ceil(rows/t)
      for i = (os-1)*t+1:os*t
```

```
        if i < rows
          var1(i)=(new_b(i) - new_A(i,:)*x_iter) / nnz(new_A(i, :));
          for j = 1: cols
            if new_A(i,j) ≠ 0
              var2(1,j)= var2(1,j)+var1(i)*new_A(i,j);
            end
          end
        end
      end
      var_div = sum(new_A((os-1)*t+1:os*t,:));
%       if i > rows
%         var_div = sum(new_A((os-1)*t+1:rows,:));
%       else
%         var_div = sum(new_A((os-1)*t+1:os*t,:));
%       end
      for j = 1: cols
        if var_div(j) ≠ 0
          var2(1,j) = var2(1,j)/var_div(1,j);
        end
      end
      % after the above for loop, var2 = sum((b_i - <A_i, x_k>/||A_i||_2^2)*A_i)
      x_iter = x_iter + weight/os * var2';

      %fprintf('%d%%...\n',floor(k/iter*100));
      %fprintf('/');
    end
  end

  tOSw(index) = toc(treconw);
  t2(index) = t2(index) + tOSw(index);
  %totaltime(index) = tReorder(index) + tOSw(index);
  %error_w = x_iter - x;
  clear new_A;
  % recon using unordered A
```

```
k = 1;
trecon = tic;
%OS-SART w/ ordering
for k = 1 : iter
  for os = 1: ceil(rows/t)
    for i = (os-1)*t+1:os*t
      if i < rows
        var1(i)=(b(i) - A(i,:)*x_iter) / nnz(A(i, :));
        for j = 1: cols
          if A(i,j) ~= 0
            var2(1,j)= var2(1,j)+var1(i)*A(i,j);
          end
        end
      end
    end
    if i > rows
      var_div = sum(A((os-1)*t+1:rows,:));
    else
      var_div = sum(A((os-1)*t+1:os*t,:));
    end
    for j = 1: cols
      if var_div(j) ~= 0
        var2(1,j) = var2(1,j)/var_div(1,j);
      end
    end
    % after the above for loop, var2 = sum((b_i - <A_i, x_k>/||A_i||_2^2)*A_i)
    x_iter = x_iter + weight/os * var2';

    %fprintf('%d%%...\n',floor(k/iter*100));
    %fprintf('-');
  end
end
clear A;
tOS(index) = toc(trecon);
```

```
  t3(index) = t3(index) + tOS(index);
  %error = x_iter - x;
%==========================
disp('finish');

%==========================
end %end for ave
t1(index) = t1(index)/5;
t2(index) = t2(index)/5;
totaltime(index) = t1(index) + t2(index);
t3(index) = t3(index)/5;
index = index+1;

end %end test_size
save('t_ordering_.mat', 't1');
save('tosw_.mat', 't2');
save('tos_.mat', 't3');
save('total.mat_', 'totaltime');



% test for full search and sum search time.
tsize = 9;
t_full = zeros(tsize,1);
t_sum = zeros(tsize,1);

index = 1;
fprintf('total round = %d.\n', tsize);
for test_size = [10 : 5 : 50]
fprintf('round %d.\n', index);


test(index) = test_size;
```

```
%-------------------------
% generate random sparse matrix A
  N = test_size * test_size;
  M = N * 3;
  %fprintf ( 'Going to generate a %4d * %4d Matrix.\n', M, N);
  i = [];
  j = [];
  v = [];
  m = M;
  n = N;

  a = sparse ( i, j, v, m, n );

  nonzeros = m * test_size;
  %fprintf(1, 'number of nonzero: %4d %4d', nonzeros, floor(nonzeros));
  for test = 1 : round(nonzeros)

    i = round ( m * rand ( ) + 0.5 );
    j = round ( n * rand ( ) + 0.5 );
    a(i,j) = 1;
    % fprintf ( 1, '  %4d  %4d  %f\n', i, j, a(i,j) );

  end

%   fprintf ( 1, '\n' );
%   fprintf ( 1, '  Number of nonzero entries is %d\n', nnz ( a ) );

  A = full ( a );
  clear a;
  new_A = zeros(m,n);
  %spy(a);
%-------------------------
% generate random matrix x
  x = (5*rand(n, 1)+5* rand(n,1))/2;
```

```
  b = A * x;

%+++++++++++++++++++++++++++
%each size loop 5 times then compute the time / 5 to get average time.
for ave = 1: 5

%+++++++++++++++++++++++++++
% full search
   [rows cols] = size(A);
  %s_r_1 = size of rays in 1 projection
  s_r_1 = test_size * 3;
  %setup a flag vector for searching projections
  num_proj = rows/s_r_1;
  for temp_i = 1:num_proj
    v_proj(1:temp_i) = 1;
  end%end for

  %new_A_p is the pointer for inserting projs into new_A
  new_A_p = 1;

  %go thru half of the projection for ref projection
  tStart = tic;
  minustime = 0;
  for ref_p = 1:num_proj
    if v_proj(ref_p)== 1
      %get the reference projection
      ref = A((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
      %flag 0 to ref proj
      v_proj(ref_p) = 0;
      %fprintf('%d ', ref_p);
      %
      minimum = s_r_1 * cols;
      %min_block =;
      for p = ref_p+1 : num_proj %p go thru cur proj
```

```
        if v_proj(p) == 1
          cur = A(((p-1) * s_r_1+1) : (p * s_r_1),:);
          % finding the overlap between ref projection and
          % current projectoin
          for i = 1:s_r_1
            for j = 1:cols
              s(i,j) =ref(i,j) && cur(i,j);
            end%end for
          end%end for
          % nnz in the S matrix is the number of overlap
          % after && operation
          sumnz = nnz(s);
          if( sumnz ≤ minimum )
            minimum = sumnz;
            min_block = p;
            min_block_i = p*s_r_1+1;
          end %end if
%       else
%         fprintf('Nothing: Reached a projection already selected.\t');
        end % end if
      end %end for
      % One full search finished
      %flag 0 to projection p
      v_proj(min_block) = 0;
    end% end if
  end% end for
%   tReorder(index) = toc(tStart) - minustime;
  t_full(index) = t_full(index)+ toc(tStart);
  %fprintf('reordering time = %f seconds.\n', tReorder(index));


%============================
% sum search
  for temp_i = 1:num_proj
```

```
  v_proj(1:temp_i) = 1;
end%end for
%new_A_p is the pointer for inserting projs into new_A
new_A_p = 1;


  %watch the time on storage the new A+++++++++++++++++
  minustime_sum= 0;
  %go thru half of the projection for ref projection
  tStartsum = tic;
  sum_A = zeros(num_proj, cols);
  for ref_p = 1:num_proj
      ref = A((ref_p-1)*s_r_1+1 : ref_p*s_r_1, :);
      sum_A(ref_p,:)= sum(ref);
  end
  for ref_p = 1:num_proj
      %fprintf('new_A_p = %d \n', new_A_p);

      if v_proj(ref_p)== 1

  %*********************************************************************
  %get the reference projection
  %flag 0 to ref proj
  v_proj(ref_p) = 0;
  %
  maximum = 0;
  for p = ref_p+1 : num_proj %p go thru cur proj
      if v_proj(p) == 1
          %%*************************

          norm1 =norm(sum_A(ref_p) - sum_A(p));
          if( norm1 ≥ maximum )
              maximum = norm1;
              min_block = p;
```

```
                min_block_i = p*s_r_1+1;
            end%end if
            %%**************************
%         else
%             fprintf('Nothing: Reached a projection already selected.\t');
        end%end if
    end%end for

    %flag 0 to projection p
    v_proj(min_block) = 0;
    %*****************************************************************
        end%end if
    end%end for
  t_sum(index) = t_sum(index)+ toc(tStartsum);



end
index = index+1;
end
save('t_sum.mat', 't_sum');
save('t_full.mat', 't_full');
plot(t_sum); hold all; plot(t_full);
```

REFERENCES

[1] Mathworks: Mathematics, sparse matrix. `http://www.mathworks.com/access/helpdesk/help/techdoc/ref/f16-5872.html`.

[2] Kenneth R. Castleman. *Digital image processing.* Prentice Hall Press, Upper Saddle River, NJ, 1996.

[3] Y. Censor. Parallel application of block-iterative methods in medical imaging and radiation therapy. *Math. Program.*, 42(2):307–323, 1988.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices.* Clarendon Press, New York, NY, 1987.

[5] A. C. Kak and M. Slaney. *Principles of computerized tomographic imaging.* IEEE, New York, NY, 1988.

[6] T. Li, Z. Liang, J. V. Singanallur, T. J. Satogata, D. C. Williams, and R. W. Schulte. Reconstruction for proton computed tomography by tracing proton trajectories: A monte carlo study. *Medical Physics*, 33(3):699–706, March 2006.

[7] S. McAllister. Efficient proton computed tomography image reconstruction using general purpose graphics processing units. Master's thesis, California State Univ., San Bernardino, CSE department, CA, Febraury 2009.

[8] S. N. Penfold, R. W. Schulte, Y. Censor, V. Bashkirov, S. Macallister, K. E. Schubert, and A. B. Rozenfeld. Block-iterative and string-averaging projection algorithms in proton computed tomography image reconstruction, 2008.

[9] K. Schubert. Keith on numerical analysis. `http://www.r2labs.org/references/KeithOnNumerical.pdf`.

[10] R. Schulte, V. Bashkirov, T. Li, Z. Liang, K. Mueller, J.n Heimann, L. Johnson, B. Keeney, H. F w. Sadrozinski, A. Seiden, D. C. Williams, L. Zhang, Z. Li, S. Peggs, T. Satogata, and C. Woody. Design of a proton computed tomography system for applications in proton radiation therapy. *IEEE Transaction on Nuclear Science*, 51(3), June 2004.

[11] A. M. Tekalp. *Digital video processing*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[12] G. Wang and M. Jiang. Ordered-subset simultaneous algebraic reconstruction techniques. *Journal of X-Ray Science and Technology*, 12(3):169–177, October 2004.