SVT Summary Report

Miki Han

# **Description of implementation:**

1. **mikiSVT()**

Inputs: n----The size of the square matrix M and also the matrix X that need to be complete

Omega ----The randomly sampled set

v ---- The data vector of the M(Omega)

delta ---- Step size

epsilon ---- Tolerance

tau ---- parameter that acts as threshold

l ---- increment

kmax ---- the maximum number of iterations

Output: Xoutput ---- The completed nxn matrix X  
Discussion: This version of the SVT uses matlab’s SVDS function with sparse matrices, and uses a precompiled .mex file to handle the sparse matrix updates. Further speed up could likely have been obtained by using a Fortran implementation of the SVD algorithm, as suggested in the paper.

**2. updateSparse()**

Inputs:

\*Y - a pointer to the sparse matrix needing to be updated

\*y - a pointer to the vector containing the updated nonzero elements of Y

Outputs:

None. This is a void function that operates on the data lying at the pointers.

Discussion: I found an example online showing how to update a matlab sparse matrix using C code, which used a loop to update the nonzero elements. I was able to improve on this by using the C memcpy() function. This allows the updated nonzero elements to be copied over as a contiguous block of bytes which is much more efficient than copying one element at a time. This resulted in a speedup of about 1.5 compared to the original version.

To compile the updateSparse.c file, matlab first needs to be linked with a compiler by calling “mex -setup”. Also, an appropriate C compiler needs to be installed, depending on the platform (mac, windows, linux).

**3. Test Script for SVT - Table 5.1**

This test script uses a for loop to call the SVT function with different values for matrix size, matrix rank, and the oversampling ratio A. The test matrixes are all randomly created, but the rank is determined. So the matrix M is created as M\_L\*M\_R’ where M\_L and M\_R are r x n to make sure the rank of the matrix. All values are stored in three arrays with the same index, and using a for loop to run all cases.

**4. Test Script for SVT - Table 5.2**

This test script uses a for loop to call the SVT function with different values for tau and step size delta. The test matrixes are all randomly created with size 1000x1000 and rank 10. All values are stored in three arrays with the same index, and using a for loop to run all cases.

**5**. **Test Script for SVT - Table 5.5**

For this part, the geodesic distance matrix data from 312 US and Canadian cities was used to test the SVT algorithm. After importing the data, calling matlab’s svds() and rank() functions show that the distance matrix has a rank of 6. In a similar method to that used for 5.1, subsets of the U and V matrices were multiplied together to create lower rank approximations of the original distance matrix. Then the SVT algorithm was ran on random subset of the original matrix. Unfortunatley, the tolerance did not steadily decrease to convergence. Tolerance decreased during initial iterations, but later increased to very large numbers.

**Results:**

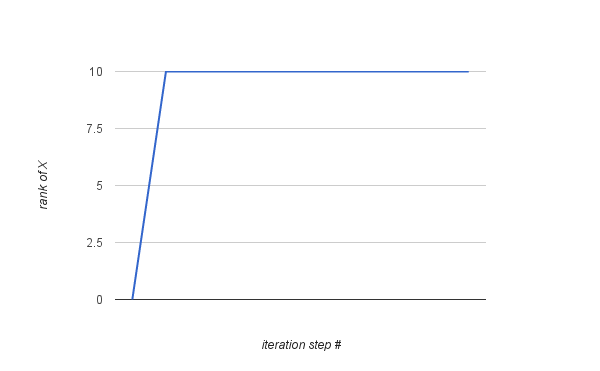
Table 5.1: Experimental results of matrix completion

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1000 x 1000 | 10 | 6 | 45 | 113 | 1.64E-04 |
| 1000 x 1000 | 50 | 4 | 204 | 114 | 1.59E-04 |
| 1000 x 1000 | 100 | 3 | 714 | 128 | 1.74E-04 |
| 5000 x 5000 | 10 | 6 | 273 | 121 | 1.55E-04 |
| 5000 x 5000 | 50 | 5 | 1743 | 108 | 1.35E-04 |
| 5000 x 5000 | 100 | 4 | 4961 | 122 | 1.45E-04 |
| 10000 x 10000 | 10 | 6 | 447 | 122 | 1.50E-04 |
| 10000 x 10000 | 50 | 5 | 1788 | 110 | 1.34E-04 |
| 10000 x 10000 | 100 | 4 | 5073 | 127 | 1.40E-04 |
| 20000 x 20000 | 10 | 6 |  |  |  |
| 20000 x 20000 | 50 | 5 |  |  |  |
| 30000 x 30000 | 10 | 6 |  |  |  |

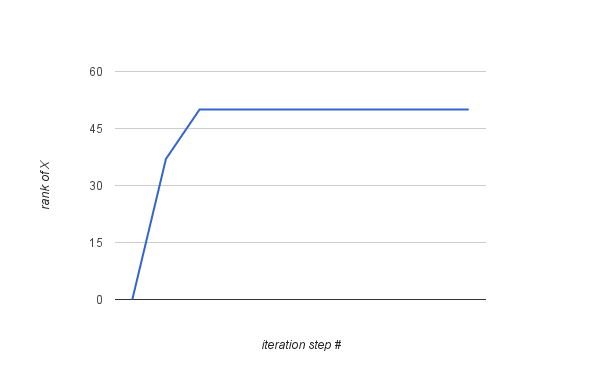
Table 5.2: Mean and Standard Deviation of rank and iteration over 5 runs

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0.8 | | | 1.2 | | | 1.6 | | |
|  | # of iterations | | rank | # of iterations | | rank | # of iterations | | rank |
|  | mean | std dev. | mean | mean | std dev. | mean | mean | std dev. | mean |
| 2n | 234 | 305.2 | 18 | 269 | 579.8 | 18 | DNC | DNC | DNC |
| 3n | 165 | 210.2 | 10 | 77 | 3.2 | 10 | 383.80 | 904.28 | 10 |
| 4n | 147 | 5.0 | 10 | 94 | 24.9 | 10 | 72.40 | 1.79 | 10 |
| 5n | 178 | 8.6 | 10 | 116 | 8.1 | 10 | 112.00 | 104.06 | 10 |
| 6n | 206 | 7.2 | 10 | 138 | 4.1 | 10 | 103.80 | 4.56 | 10 |

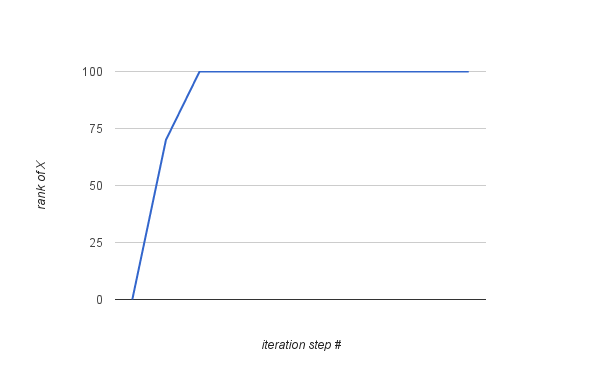
Fig 5.1: Rank vs. Iteration



Part A: Testing mikiSVT with n=5000, rank=10, and A=6



Part B: Testing mikiSVT with n=5000, rank=50, and A=5



Part C: Testing mikiSVT with n=5000, rank=100, and A=4

**Discussion of Results:**

My experimental results were closely aligned to those in the paper. Some differences may have been caused by differences in the computers used. I initially ran my code on windows 7 laptop containing a dual-core Pentium CPU and 4 GB of RAM, but I encountered ‘out of memory’ errors as I started to testing 10000 x 10000 random matrices. Increasing the memory swap size to 8GB fixed this error, but then the algorithm started taking a very, very long time, perhaps because the matrix was being written to and from the hard drive during every iteration. The code ran much faster (approximately 3x faster) after switching to a 6-core desktop computer with 16GB of RAM, but eventually this computer crashed as well. Then I decided to run the code on some of the machines in the computer lab. This was much faster since I could log onto multiple machines , but the memory swap size was limited to 1GB, and cannot be changed without administrative privileges. Overall, the SVT algorithm worked well and fairly effeciently.