ECSE 444 Final Project

Matrix Solver: SVD Implementation and Optimization

Yinuo Wang, 260766084

Abstract

The scope of this report includes a brief introduction regrading to Singular Value Decomposition (SVD) algorithm used as a matrix solver, including the procedure of a specific example and optimization. This algorithm constitutes one section of the entire group project, thus comparison with other programs will also be included. A conclusion reached is that SVD serves as an intuitive method, but not an efficient and straightforward one. Within the current mathematical scope we have, implementing a general stand-alone SVD program is overreaching in algorithmic logics.

I. Introduction

Since in real life, large chunks of data are often encoded as matrices, solving a linear system are usually demanding, meaning numerical operations in enormous scale must be modified. Inside of the project groups, other members have probed into algorithms like LU and QR decompositions targeting at matrices of different attributes, like symmetric positive definite or good conditioned. SVD decomposition, unlike those algorithms mentioned above, focus on generalizing the inversion of matrices on different systems (over and under constrained). The generate form of SVD of a rectangular matrix A is shown below in figure 1.

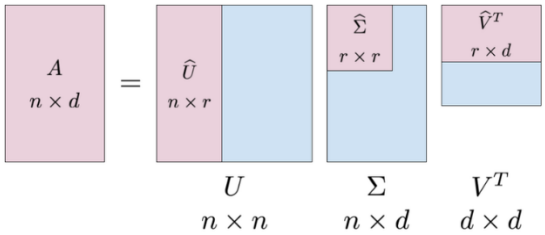

Figure 1: A illustration of SVD components

U and V are matrices containing orthonormal vectors. V is the normalized eigen matrix of A^TA and U can be calculated from $A \cdot V$. Σ is a diagonal matrix having non-zero elements as the square root of the original eigen values of A^TA. To sum up, what SVD does is that it counters the difficulty of the inversing non-square matrices. By laying out SVD components, inversed rectangular matrix is thus represented, and solving the system becomes a series of multiplications, shown in equations below. Note the orthogonality of U and V vectors circumvent the inverse into transpose and inverse of a diagonal matrix Σ^* is simply taking reciprocal onto every element.

$$(U\Sigma V^T)x = b$$
$$x = V\Sigma^* U^T b$$

II. Procedure

Two technical bottleneck limits the exploration of SVD and thus the resulting algorithm makes a compromise on generality to demonstrate that solving linear system using SVD is feasible. First, SVD applies to any matrix at any shape. It is difficult to capture all those possibilities using C language. The designed algorithm only handles square ones. Another difficulty is that a robust eigen decomposition algorithm can not be found. There is a routine found in book "Numeric Recipes in C" at chapter 11, and attempts are made to use the canned methods. However, it gives "segmentation fault" during execution. An alternative solution is to use python routine, which breaks the overall integrity of the algorithm. Below list a flow chart describing the algorithm. The aborted "eigen.c" and "nrutil.c" source codes are both included in the submission.

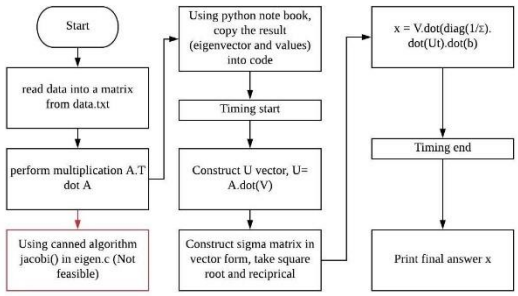

Figure 2: A flow chart of the overall logic of SVD solver

Since using SVD to solve linear systems are matrix multiplications essentially, components

in the final implementation are straightforward. Routine `double** dotTranspose(A, size)` encapsulates the transpose and dot multiplication together and complete the preparation before eigen. After the aid from python, the following operations are all packed into a `double* SVDSolver(size)` method which undertakes the construction of U and modified Σ, and complete the cascading multiplications. In the initial version of SVD algorithm, all multiplications are naïve iteration. All datatype used are `double`. Every matrix returned are dynamically allocated 2D arrays.

A conclusion reached by using a specified 5-by-5 matrix reached is that the entire algorithm is proved working by returning the correcting result (as figure 3 shows below), but the injection of python result inside the code makes the algorithm encumbered with low testability.

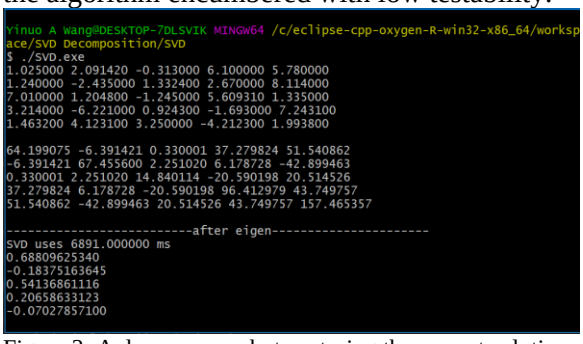

Figure 3: A demo screenshot capturing the correct solution vector (numbers under timing).

III. Optimization

The optimization follows an incremental pattern where implementations are modified step by step. Below list the optimizations and result, along with a brief conclusion. For more details please refer to the source files. All the timing data can be found in the spread sheet submitted with the source file.

In common with other sections of the group, the timing analysis uses the time.h header file. An example 5-by-5 matrix is timed after eigen decomposition calculation and timed with 100000 iterations.

Original SVD: 6841ms (with fluctuations)
Float SVD: This implementation replaces all double types into float and added specific cast at certain operations such as taking square root. It takes 6463ms.

Float SVD with memory handling: Building on the float calculation, this optimization target on the naïve allocation which happens whenever a new matrix pointer needs to be returned. At the end of every iteration, all allocated memory will be recycled. It takes 6471ms.

Float SVD with memory handling and vectorized: This is the ultimate version. Inspired by LU and QR algorithms where the matrix is represented in long vector of size n^2 rather than n-by-n 2D array. Significantly faster, 100000 loops now take 5657ms.

To sum up, changing the double into float type only boosted the performance to a small degree. An anticipation is that operations like square root still requires multiple cycles. Explicit castings probably increase the overhead too. For memory management, it does not perform any modification in speed since more instructions are added every iteration. The computer memory might be sufficient to hold 100000 arbitrary allocations originally. What really increase the speed of the algorithm is when the matrix is pulled and flattened into a long 1D array, the program now has much simpler memory allocating processes. The traversal of the elements is also more convenient.

IV. Conclusion

SVD components are proven capable of solving a linear system. Several optimizations demonstrate its potential. However, comparing to LU or QR decomposition, the cascaded matrix multiplication is of a few hundred times slower for a matrix of 5-by-5 according to the data. One possible way of interpreting this is that matrix multiplication is of $O(n^3)$. SVD makes the understanding of the solving process easier by transferring inverse into a series of dot products, but computationally much more complicate. In real life SVD are handled in more elegant ways instead of the deductive-style implementation in this project. Also, advanced algorithms realize the generality of SVD, that is, they can handle matrices of random shapes.