Report - Linear system of equations solvers

ACSE5 – Advanced Programming Assignment
Gabriel Lipkowitz, Albert Celma Ortega, Nicolas Trinephi
February 2, 2020

Report overview

- Approach
- Core concepts
- Implementation
- Method analysis: Performance
- Takeaways on strategy and software design

1. Approach

Solving complex linear systems of equations quickly and efficiently represents an increasing field of research, and its applications have rapidly grown in the last decades jointly with computational capacity improvements. We decided to first develop a set of iterative methods for solving dense, sparse and banded matrices. From there, we decided to expand into direct methods for dense matrices and finally developed the adaptation for sparse matrices (LU decomposition and Cholesky methods).

2. Core concepts

Each of our methods were built in the developed Matrix, CSRMatrix, and BandedMatrix classes, declaring methods in the respective header files. For testing purposes, to ensure that our solvers indeed solve an $A\mathbf{x} = \mathbf{b}$ type system of linear equations, we included a Matrix_Tester class and its own corresponding .cpp and .h files. A first important programming technique for our implementation was *inheritance*. As subclasses of the dense Matrix class, the sparse and banded matrix classes inherited all the properties of a generic dense matrix, such as public variables like columns and rows. *Polymorphism* was also of crucial importance. Namely, several functions in the dense, sparse, and banded matrix classes completed the same operation (e.g. matrix column-vector multiplication, matrix row-vector multiplication, sparse and dense matrix-matrix multiplication, along with the Jacobi, Gauss-Seidel, SOR, conjugate gradient, and Cholesky solvers). Thanks to polymorphism, when we called these functions, our programme could automatically implement the correct method corresponding to each class. Finally, all our functions, for all three of our matrix classes, were templated so that the matrices can contain integers, doubles, floats, etc.

3. Implementation

While we initially tested our solvers with simple matrices generated manually, we also wrote functions to randomly generate dense, sparse, and banded matrices, of different sizes, and sparsities and bandwidths, if applicable. Sparse matrices were stored in the old Yale format, whereas banded matrices were stored as an *n x b* array, where n is the number of rows and b is the bandwidth. Matrices could be filled randomly such that they were diagonally dominant (see iterative solvers' requirements for convergence, below) and/or symmetric positive definite (see conjugate gradient solver's requirements, below).

The simple iterative solvers – Jacobi, Gauss-Seidel, and Successive Over-Relaxation (SOR) – all can converge to the solution for diagonally dominant matrices. Our matrix generation functions, described above, accomplished this by enforcing the diagonal elements to be artificially large.

The conjugate gradient method requires, for convergence, that the matrix be symmetric positive definite (SPD), which is defined by the following equation:

$$\mathbf{z}^T A \mathbf{z} > 0$$

To generate an SPD matrix, as a function we wrote did, we first generated a lower triangular matrix, with nonzero diagonal elements, and then multiplied it by its transpose, an upper triangular matrix also with nonzero diagonal elements. To perform these operations, we implemented subfunctions including matrix transposition and matrix-matrix multiplication, for both sparse and dense formats.

The algorithm for sparse matrix-matrix multiplication, taken from Bank and Douglas (2001) [1], involved first determining the sparsity structure of the matrix product, and then filling it with the appropriate values.

Sparsity

<u>-Comments on LU sparsity: index looping complexity</u>

LU decomposition algorithm structure

1. Gaussian elimination & Lower triangular storage
    1. Diagonal element
    2. Lambda factors for different non-zero entries in given column
    3. Elimination of non-zero entries in given column
    4. Row entries **update*** doing same row operation to make zero in column(using lambda)
    5. Storage of lambda in lower triangular matrix L.
2. Forward and Backward substitution

LU gaussian elimination requires linking the information provided by each of the vectors from CSRMatrix *class* to do the steps above. The greatest complexity is in updating the row entries and dealing with new non-zero entries appearing because of the row operation. The way to solve this is allocating enough memory at the beginning. This could be done symbolically by doing the operations and with a counter assessing number of new entries or setting a larger initial "nnz" values and operate to update the information within the vectors ("nnz", "row_pos", "col") as it goes eliminating non-zero entries under the diagonal at each column. The latter approach was used for time constraints given that its suboptimal yet better than creating copies. The hardest task comes once the elements under the diagonal are eliminated and then updating the row values new non-zero entries appear and have to be allocated in the correct index position at the "nnzs" vector. Which is solved creating an update of the values vector similar how a *push_back* would in a vector. This is complex because it needs to be ordered by columns therefore must check the "nnz" entries and allocate itself in the right place. Finally, storing the elements in a Lower matrix requires to save the *alpha* or mult value found on the *CSRMatrix* L used.

<u>-Comments on Cholesky:</u>

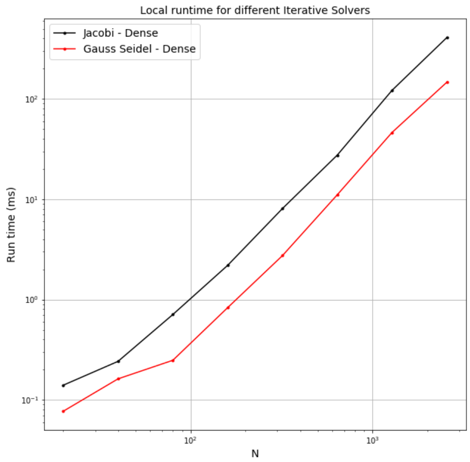
The Cholesky Factorization is like the LU decomposition in that they both seek to create a lower triangular matrix with non-zeros that do not correspond to the original matrix. Each non-zero of the lower triangular portion of A will be a non-zero in L, ignoring mathematical cancelling (if the s value indeed zeroes itself). However, there is a link between these non-zeros in L when looking for fill-in values, these are values that are non-zero in L but correspond to a zero value in A. For $i < j < k$, if both l_{ij} and l_{ki} are non-zero then l_{kj} is non-zero [S. Parter, 1961], as such finding the fill-ins of L is possible but finding its position within a CSR matrix is complex. First, the aim is to find the structure of the sparse L, then to fill it with the coefficients and then to transpose it and solve with back and forward substitution. Time permitted for only the structure to be found, for now.

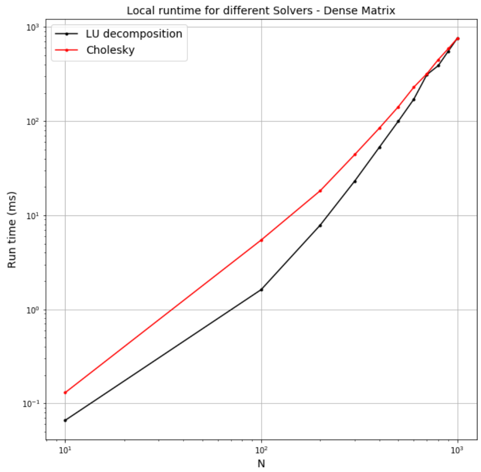
Page Break

4. Performance analysis

As expected, Gauss-Seidel, which takes advantage of more information than does Jacobi, both requires fewer iterations to converge to a solution, and takes less time to do so.

n (square matrix size)	10	20	40	80	160	320	640	1280	2560
Jacobi method	16	15	16	16	16	16	15	15	14
Gauss-Seidel method	6	6	6	6	6	6	6	6	5





For dense matrices, as shown below, the Cholesky method demonstrated *$O(n^2)$* run time, i.e. as the matrix size increased by a factor of 10, run time increased by a factor of 100.

Description of LU Factorization:

Case 1

$$\begin{bmatrix}5&3&0&0\\0&2&0&1\\1&0&4&0\\0&3&0&1\end{bmatrix}\rightarrow\begin{bmatrix}5&3&0&0\\0&2&0&1\\0&0&4&0\\0&3&0&1\end{bmatrix}\rightarrow\begin{bmatrix}5&3&0&0\\0&2&0&1\\0&0&4&0\\0&0&0&0\end{bmatrix}LU^{(2/2)}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\\mathbf{0.2}&0&1&0\\0&\mathbf{1.5}&0&1\end{bmatrix}\begin{bmatrix}5&3&0&0\\0&2&0&1\\0&0&4&0\\0&0&0&-0.5\end{bmatrix}\ \mathit{Correct}$$

Case 2

$$\begin{bmatrix}5&0&3&0\\0&2&0&1\\1&0&4&0\\5&3&0&1\end{bmatrix}\rightarrow\begin{bmatrix}5&0&3&0\\0&2&0&1\\0&0&3.4&0\\0&3&-3&1\end{bmatrix}\rightarrow(*)\begin{bmatrix}5&3&0&0\\0&2&0&1\\0&0&4&0\\0&0&-3&-0.5\end{bmatrix}L=LU^{\left(\frac{1}{2}\right)**}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\\mathbf{0.2}&0&1&0\\\mathbf{1}&\mathbf{0}&0&1\end{bmatrix}\begin{bmatrix}5&3&0&0\\0&2&0&1\\0&0&4&0\\0&0&0&-0.5\end{bmatrix}\ \mathit{Partial}$$

*Appears non zero entries and this represents an update of the *sparse matrix* vectors
** second iteration not reached completely correctly

1. Areas for improvement

1. Mathematical models

We feel happy and proud of the broad range of methods that we have built. We consider that we had the right approach focusing on iterative methods and then building direct methods. With respect to direct methods, we covered LU decomposition, and Cholesky. These two are a good sample of direct methods. We feel that although significant improvement was made for LU decomposition as well as Cholesky for sparse matrix notation, we should have had a different strategy to be more efficient in managing time.

2. Effective team work

The most important takeaway from this course is definitively how to allocate work effectively to optimise our output. We decided to run on parallel for LU decomposition sparse as well as Cholesky thinking that we could work together having both solvers done and then work to get forward and backwards substitution done. But Cholesky and LU decomposition implementation were further apart than the actual concepts had allowed us to foresee. These methods for sparse matrices are incomplete but a good effort was put into trying to complete them. The take away is to work parallel only if tasks are not superimposable, or said differently, that once one both are finished and assembled, you can have a working piece of software.

References:

S. Parter, *The use of linear graphs in Gauss Elimination,* SIAM Rev., 3 (1961), pp. 119-130.

Bank, Randolph and Douglas, Craig (1993) "Sparse Matrix Multiplication Package (SMMP)" *Advances in Computational Mathematics* **1**.