ACSE-5 Project 3 - Positive Definite Matrix Solver

Team Mr Fantabolous: Richard Boyne

Overview

The task set was to write a solver for the matrix equation:

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

where **A** is a square Positive Definite (PD) matrix, **b** is a known vector and **x** is an unknown vector. In general iterative solvers are more versatile and easier to implement over direct methods[1]. The Jacobi method is the simplest of these methods, here Gauss Seidel is used as it is an improvement with little extra complications. Rather than implementing different methods here the aim of this code was to compare different matrix storage methods performance, namely full 2D array form (dense) and Compressed Sparse Row (CSR) form.

A sample system is needed to test the solver. Diagonally dominant matrices are generated (to ensured Positive Definiteness) with random off-diagonal entries and a fixed sparsity (see readme for more specifics). The a vector **x** is also randomly generated, then used to find **b**, thus setting up the problem.

Code Features

Use of a pure virtual matrix class ensures that each for of matrix meets a bare minimum of functionality (specifically one must be able to set and index values) as well as enabling basic functions such as print_matrix to be inherited.

Code is designed for user-developers, hence all class members are made public under the assumption that the user knows what they are doing. This reduces the robustness of the code but reduces it's complexity.

Unique smart pointers are used to assist with dynamic memory handling, meaning that memory is automatically deleted when the matrix goes out of scope. This is reasonable as once the matrix goes out of scope it is unlikely a user would still want to access its memory.

Templates are used for both matrix types (sparse and dense) to allow for different data types, however here only doubles are used for example problems.

Using the Code

To use this code one needs to initialise a class instance for the matrix type required, then call the set method for every element to be given a value (refer to readme for the caveats of the various set methods). The vector **b** must then be setup as a dense matrix with 1 column. To solve the Gauss_Seidel method of the matrix must be called passing the number of iterations to run for. A set of examples are shown in "main.cpp" with a user interface to run them upon compilation.

Performance

Generated matrices with varied size and supercity were solved and averaged over at least 10 different random seeds for both matrix types.

Accuracy

Over 100 iterations results were found within a standard deviation of $\mathcal{O}(10^{-13})$ or better for matrices up to 5120×5120. However diagonally dominant matrices are more optimal for this kind of iterative solver. Testing with other types of PD matrices is required for ensuring this level of accuracy is reasonable for all PD matrices.

Code Optimisation

To improve performance the loops for both matrix types have their complexity reduced, specifically not calling the index functions was key to improve run time. They also access stored data in a contiguous manner so as to optimise cache memory. Run time comparison of CSR to dense matrix algorithms is shown below:

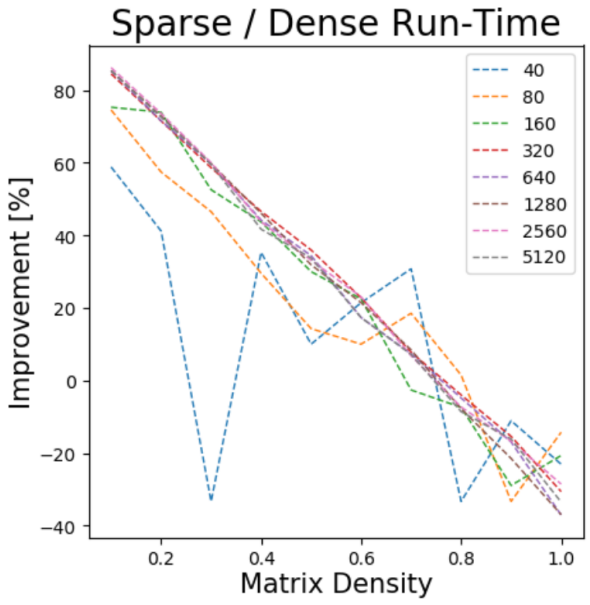

Fig. 1. Run time difference of CSR over dense matrix storage as a percentage of dense method run time for various sparsity's and sizes.

For small matrices the run time becomes so small that improvements are either non-existent or can't be measured, leading to seeming random results.

for matrices with size above $\sim \mathcal{O}(10^2)$ there is a relatively constant trend of improvement up to 80% for densities of 0.1. More significantly matrices more dense than 0.8 are slower

for CSR than dense. This is due to the increased overhead and memory needed to be accessed for a full CSR matrix (it is also worth noting that over double the storage is required). This is an expected trend, thus indicates the algorithms have been implemented correctly.

Compiler Flag Optimisation

Turning on the Ox compiler flag allows for improved batch processing of non-interacting loops that the compiler identifies. On the 5120 matrix this gives an average run time improvement of 54.1% for dense matrices and 63.7% for sparse matrices. This would likely be improved further with use of BLAS or LAPACK routines.

Weaknesses and Improvements

Though the code is reasonably quick and accurate for the problems tested there are still many areas that could be improved:

- Convergence criteria has not been inbuilt into the solver, meaning the user must estimate the number of iterations needed for a given accuracy.
- Only one method of solving is provided, which may not be optimal for many PD problems. Implementation of LU decomposition would be good as many **b** vectors can be solved with for reduced computation after the first. This would also be a direct method removing the need for convergence criteria.
- Code needs to be validated for other types of PD matrices. Also it would be good to consider methods that can solve non PD problems (e.g. approximating a similar PD to the NPD problem).
- Publicly accessible properties variables for an increasing complex class is dangerous as even the most experience programmers can break the inner workings accidentally. Given more time I would make more properties of these classes protected to increase robustness.
- Speed could be optimised further by building on lower level BLAS or LAPACK routines. However, at some point one would just be recreating the higher level routines in the same package.

References

[1] William L. Briggs, Van Emden Henson, and Steve F. McCormick. 2000. A Multigrid Tutorial: Second Edition. Soc. for Industrial and Applied Math., Philadelphia, PA, USA.