CME 338 Final Project: USYMLQ and USYMQR

Jui-Hsien Wang

September 5, 2017

1 Introduction

In this project, we implemented the USYMLQ and USYMQR algorithms for solving large, sparse linear systems in templated C++. The algorithms are defined in the paper by Saunders *et al.* [1], and in a closely-related paper by Reichel *et al.* [2]. These two methods were seen as extension to the Conjugate Gradient (CG) and the Minimum Residual (MINRES) methods for unsymmetric matrices. In addition, as pointed out in [2], USYMQR (or in their paper, "generalized LSQR", despite there is no special case in which USYMQR reduces to LSQR) can handle least-square problems (both of them can solve underdetermined system).

Our main contribution, in addition to providing a tested implementation in yet another language, is to evaluate the algorithms with a variety of matrices, including *The University of Florida Sparse Matrix Collection* by Tim Davis *et al.* [3]. About a thousand different matrices originated from problems in many different areas (please see [3] for the collection) were run through the USYMLQ and USYMQR solvers. In addition, for a few selected problems, we compared the algorithms to the perhaps more well-known sparse linear system solvers including BiCGStab, LSQR and LSMR. Since these implementations are only available to the author in Matlab, the comparison will be based on the number of iterations given similar error tolerance for the same problem.

2 Tridiagonalization

Given an unsymmetric, real $M \times N$ matrix A, the following decomposition exists (please see [1] for proof)

$$P^* A Q = T, \tag{1}$$

where P and Q are orthogonal matrices and T is a tridiagonal matrix. Note that if A is symmetric then we can choose $P = Q$, then T is symmetric and the symmetric Lanczos process can be derived. This decomposition is unique given a pair of starting vectors $p_1 \in \mathbb{R}^M$ and $q_1 \in \mathbb{R}^N$, which are the first columns of P and Q, respectively. The algorithms can be easily extended to complex matrices, and focus will be given to only real matrices from now on.

At step n, let $P = P_n = (p_1, p_2, \cdots, p_n)$, $Q = Q_n = (q_1, q_2, \cdots, q_n)$, and

$$T = T_n = \begin{bmatrix} \alpha_1 & \gamma_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \gamma_3 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \beta_{n-1} & \alpha_{n-1} & \gamma_n \\ 0 & \cdots & 0 & \beta_n & \alpha_n \end{bmatrix}. \tag{2}$$

With this notation, we can derive an iterative tridiagonalization algorithm TriDiag following [1]. Instead of a three-term recurrence relation in Lanczos process for symmetric matrices, we get two three-term recurrence relations. When A is symmetric, this algorithm falls back to the Lanczos process. Note that this is not a

Krylov-subspace method, but an analogy in the sense that the union of subspaces span(Q_{2n}) and span(Q_{2n+1}) contains the Krylov subspace generated by n steps of the symmetric Lanczos algorithm applied to the normal equations. In addition, they contain the space spanned by the intermediate vectors obtained if the matrix of the normal equations is not formed explicitly, but the multiplication by A and A^* are done in sequence [1].

Algorithm 1 Tridiagonalization of an Unsymmetric Matrix.

Require: A, and two arbitrary vectors $b \neq 0$, $c \neq 0$.

```
function TriDiag(A, b, c).
    β_1 = ‖b‖, γ_1 = ‖c‖;
    p_0 = q_0 = 0, p_1 = b/β_1, q_1 = c/γ_1;
    for i = 1, 2, 3, ... do
        u = A q_i − γ_i p_{i−1};
        v = A^* p_i − β_i q_{i−1};
        α_i = p_i^* u;
        u− = α_i p_i;
        v− = α_i^* q_i;
        β_{i+1} = ‖u‖;
        γ_{i+1} = ‖v‖;
        if β_{i+1} = 0 or γ_{i+1} = 0 then
            break ;
        else
            p_{i+1} = u/β_{i+1};
            q_{i+1} = v/γ_{i+1};
        end if
    end for
end function
```

3 Solving Linear Systems

The algorithm TriDiag can be used to solve the linear system

$$Ax = b, \tag{3}$$

where A is a general, unsymmetric $M \times N$ matrix. When $M > N$, we seek the solution to the least-square problem $\min_x \|Ax - b\|_2$. Note that in the tridiagonalization algorithm, A is only defined by the operations $y \leftarrow Ax$ and $y \leftarrow A^*x$. Therefore both algorithms are suited for solving large, sparse systems.

We start the algorithm by assigning $p_1 = r_0/\|r_0\|$, where $r_0 = b - Ax_0$ is the residual for an initial guess x_0. q_1 can be chosen randomly. However, in [2], it was shown that it can be beneficial to set $q_1 \approx x$, which is generally unknown except in certain applications where estimation is possible. After j steps of TriDiag, we can approximate the solution to Eq. (3) in the affine subspace $x_0 + \text{span}(Q_j)$. Depending on the desired properties of the residuals at step j, $r_j = b - Ax_j$, we have two types of methods: USYMLQ and USYMQR The following descriptions closely follow the derivation in [1].

3.1 USYMLQ

We denote x_j^{cg} the solution given by USYMLQ such that the residual vector $r_j^{cg} = b - Ax_j^{cg}$ is orthogonal to span(P_j). This is an oblique projection method and x_j^{cg} can be computed by:

$$\text{Solve the } j \times j \text{ tridiagonal linear system } T_j h_j^{cg} = \beta_1 e_1. \tag{4}$$

$$\text{Set } x_j^{cg} = x_0 + Q_j h_j^{cg}. \tag{5}$$

An efficient implementation based on LQ factorization of T_j can be found in [1], using only two M-vectors and four N-vectors. Updates at each step has complexity $\mathcal{O}(MN)$ due to matrix-vector multiplications.

3.2 USYMQR

We denote x_j^{cr} the solution given by USYMQR such that the residual vector $r_j^{cr} = b - Ax_j^{cr}$ is minimal in the sense that $\|r_j^{cr}\| = \min_q \|b - A(x_0 + q)\|$, $q \in \text{span}(Q_j)$. This implies the monoticity of $\|r_{j+1}^{cr}\| \leq \|r_j^{cr}\|$. In fact this method reduces to MINRES when A is symmetric, hence the name conjugate residual method. x_j^{cr} can be computed by:

$$\text{Solve the } (j+1) \times j \text{ least-square problem } \min_{h_j^{cr}} \|S_j h_j^{cr} - \beta_1 e_1\|. \tag{6}$$

$$\text{Set } x_j^{cr} = x_0 + Q_j h_j^{cr}, \tag{7}$$

where

$$S_j = \begin{bmatrix} T_j \\ \beta_{j+1} e_j^* \end{bmatrix}. \tag{8}$$

An efficient implementation based on QR factorization of S_j can be found in [1], One extra N-vector of storage is needed compared to USYMLQ.

4 Implementation and Solver Design

The above algorithms were implemented in single-threaded C++. The implementation is abstracted and templated so different numerical linear algebra library can be used as long as it supports (1) matrix-vector/scalar-vector multiplication; (2) matrix transpose; (3) dot product, and (4) advanced initialization such as initializing zeros or random numbers (implemented as static methods). We used Eigen, a free and open-sourced C++ template library for linear algebra [4]. It is header-only and the source code of version 3.3.4 is included in the repository. Our USYMLQ implementation was compared to a Matlab version and on average is 30% slower, which is probably due to the underlying matrix-vector multiplication performance difference. Matlab, which uses Intel MKL as backend, performs matrix-vector multiplication about 30% faster than Eigen; this test was performed on a 2000×2000 matrix (0.0024 sec vs 0.0036 sec). Although in the Eigen benchmark tests [5], it shows that for matrix-vector multiplication Eigen is faster than MKL. This does not seem to be the case given my clang compiler on Mac with O3 optimization.

Use of the solver is simple. Following is a code snippet for how to use the solver given A and b. It is easily configurable for solve mode (`USYMLQ` or `USYMQR`), logging output (e.g. `std::cout`), logging verbose level, maximum timestep, and solver tolerance. In the snippet, `T_Vector` and `T_Matrix` are the template input arguments.

```
/* --------- SOLVER EXAMPLE --------- */
// ... define A and b and x0 ...
T_Vector x;
T rnorm;
USYM_Linear_Solver<T,T_Vector,T_Matrix> solver(A,b);
solver.Initialize(x0);
solver.Set_Mode(USYMQR);
solver.SetMaxIteration(2000);
solver.Set_Tol(1E-12, 1E-12);
solver.Solve(x, rnorm);
```

Termination criteria The solver returns an integer flag at termination to inform the user about the result of the solve. It is summarized in the following table.

Stop flag	Reason for termination
0	$x = 0$ is the exact solution. No iterations were performed
1	$\|Ax - b\|$ is sufficiently small, given the values of a_{tol} and b_{tol}.
2	$\|A^*(Ax - b)\|$ is sufficiently small, given the values of a_{tol}.
4	$\|Ax - b\|$ is as small as seems reasonable.
7	The iteration limit `maxItn` was reached.
11	The system $Ax = b$ is incompatible given b_{tol}.

The notion of "sufficiently small" is defined relatively to the problem. In general, the error tolerance is defined by two user-specified numbers, a_{tol} and b_{tol}. All the matrix norms defined below are Frobenius norms; all the vector norms are Euclidean norms. We make a further approximation that at step j, the norm of A can be approximated by $\|A\| \approx \|T\|$. The advantages of this approximation are that (1) its usually cheaper than running through the entire matrix ($\mathcal{O}(j)$ compared to $\mathcal{O}(MN)$ although the latter can be computed only once), and (2) $\|T_{j+1}\| \geq \|T_j\|$; if they are equal then we find the exact solution because TriDiag terminates, if they are not equal then $\|T_j\|$ is increasing monotonically, which implies monotonically increasing solver tolerance (see below) and higher likelihood of termination. We explain the triggering conditions for each flag in the following paragraphs.

`Stop flag 1`:

$$\frac{\|r\|}{\|b\|} < b_{tol} + a_{tol}\frac{\|A\|\|x\|}{\|b\|}, \tag{9}$$

`Stop flag 2` (only in least-square problems):

$$\frac{\|A^*(Ax - b)\|}{\|A\|\|r\|} < a_{tol} \tag{10}$$

`Stop flag 11` (only in least-square problems):

$$\frac{1}{\|r\|} < b_{tol} \tag{11}$$

5 Results

We compare our USYMLQ and USYMQR implementations to the widely used LSQR, LSMR and BiCGStab algorithms. LSQR and LSMR are suitable for solving both square and rectangular systems (including under- and over-constrained systems), and BiCGStab is suitable for square system. We used the Matlab built-in version of LSQR and BiCGStab, and community supported Matlab implementation of LSMR [6]. Because our implementation is in C++, it does not make much sense to compare the runtime speed across these algorithms. Instead, our comparison will be based on number of iterations mainly since all of these algorithms have matrix-vector multiplications at every step, which should be the main bottleneck for speed.

Some numerical experiments comparing a subset of these algorithms can be found in the literatures. In [1], a family of special block tridiagonal matrix was constructed to compared USYMLQ and USYMQR to LSQR. This 1-parameter matrix family has the following shape,

$$A = \begin{bmatrix} B & -I & & & \\ -I & B & -I & & \\ & \ddots & \ddots & \ddots & \\ & & -I & B & -I \\ & & & -I & B \end{bmatrix}, \quad B = \begin{bmatrix} 4 & a & & & \\ b & 4 & a & & \\ & \ddots & \ddots & \ddots & \\ & & b & 4 & a \\ & & & b & 4 \end{bmatrix}, \tag{12}$$

where $a = -1 + \delta$, and $b = -1 - \delta$. It was found that in general LSQR converges in less iterations when $\delta \geq 1.0$. In [2], it was found that USYMQR converged faster than LSQR when the solution cannot be well approximated by vectors in low-dimensional Krylov subspaces, for example, a near-constant function given by the discretization of Fredholm integral equation of the first kind. Furthermore, [2] observed that it can be beneficial to choose the second starting vector in the tridiagonalization, q_1, so that it approximates the solution, $q_1 \approx x/\|x\|$. The use of this was demonstrated in an image deblurring problem, where $b \approx x$ and therefore was available. However, it is generally not true one would have an approximate x before the start of the algorithm. Although these tests were well-designed to probe some properties of the algorithm, the selection of problems can cause bias in the evaluation. In the following sections, we tested USYMLQ and USYMQR on (1) dense random matrices and (2) the UFL sparse matrix collection in an attempt to gain more insights about the algorithms.

5.1 Dense, random matrices

The first set of test cases are square matrices that are dense and initialized randomly. The entries of the matrix are sampled uniformly between $[-1, 1]$ through Eigen's advanced initialization methods. Right-hand-side vector b is also initialized randomly and have values between $[-1, 1]$. Matrices of three different sizes were initialized: 50×50, 500×500, and 5000×5000.

Both USYMLQ, USYMQR determine the exact solution in at most N steps (see [1] for proof) in exact arithmetics, however, numerical errors cause the matrices P and Q to loss orthogonality after some steps and in general equation (1) might not hold. Krylov subspace methods all have similar properties. It is therefore not surprising to see that the number of iterations needed is higher than the dimension of the problem, after which the residual drops rapidly to suggest a subspace containing the solution has been found. However, it is surprising to see that both USYMLQ and USYMQR found this space a bit earlier compared to LSQR and LSMR, suggesting that they might be more robust against orthogonality drift. In general, we found that USYMQR has much better convergence properties than USYMLQ, which oscillates quite a bit and cannot be stopped early. USYMQR, due to its minimal residual guarantee, $\|r_{j+1}^{cr}\| \leq \|r_j^{cr}\|$, has smoother convergence and can be stopped early for higher error tolerance. Please see figure 1-3 for the comparison.

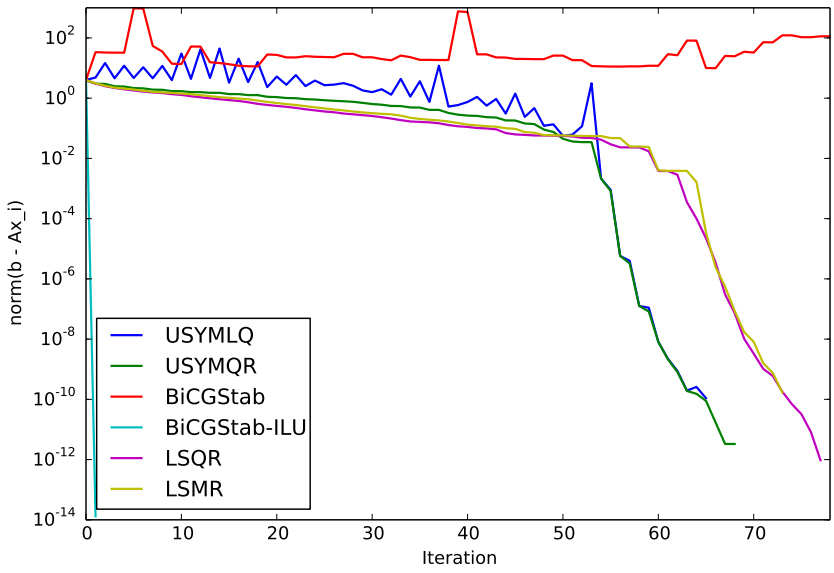

Figure 1: $M = 50$, $N = 50$, A and b are randomly initialized.

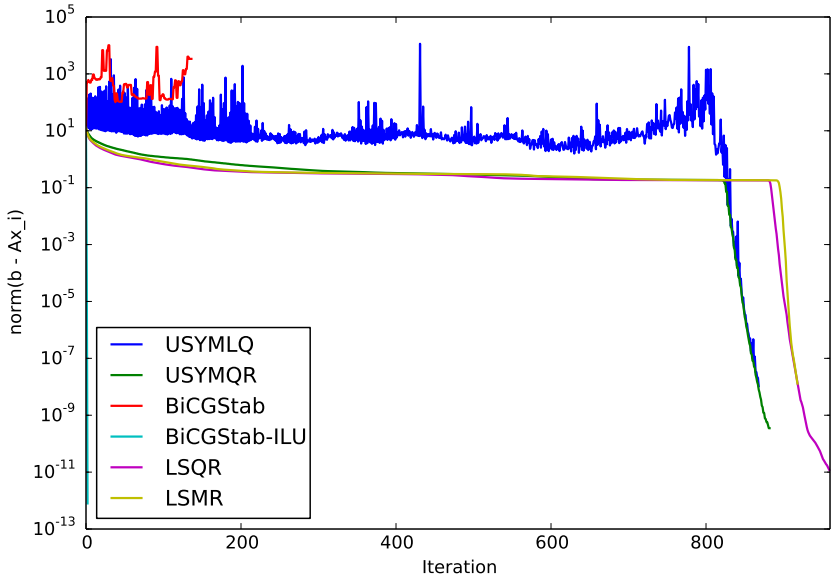

Figure 2: $M = 500$, $N = 500$, A and b are randomly initialized.

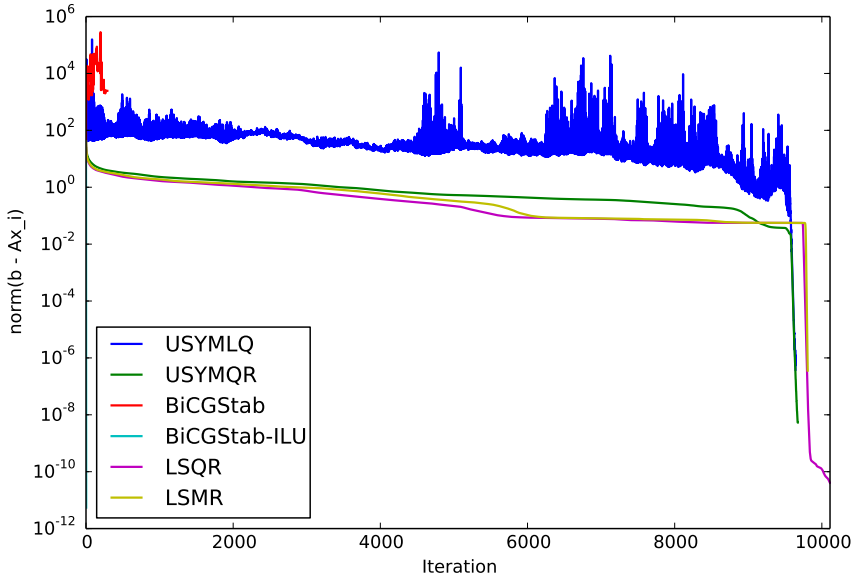

Figure 3: $M = 5000$, $N = 5000$, A and b are randomly initialized.

Next we look at three different random least-square problems ($M > N$) with 50×25, 500×250, and 5000×2500 dimension. The same randomization as square systems was applied to generate A and b. We found that for compatible systems ($Ax = b$), all algorithms converged quickly with iteration count $j < N$ (see figures 4-6). Except for the smallest test case ($M = 50$, $N = 25$), LSQR and LSMR converged faster than USYMQR and USYMLQ. Like square systems, USYMQR demonstrates smoother convergence behavior than USYMLQ. For incompatible systems ($Ax \approx b$), we exclude USYMLQ because it is not suitable for incompatible least-square problems. For incompatible system, we quantify the convergence using the relative

least-square error $r_{lsq} = \|A^*(Ax - b)\|/\|A\|$. The results are shown in figures 7-9. We observed that there seem to be an implementation-related issue to prevent the USYMQR solver from converging below 10^{-8}, and therefore our stopping criteria ($a_{tol} = b_{tol} = 10^{-12}$) was never triggered (see paragraph "Termination criteria" for descriptions). Because USYMQR only guarantees monotonically decreasing $\|r\| = \|Ax_j - b\|$, we see that in all three cases the least-square error actually increased. Interestingly, this seemingly implementation error does not appear in either square or underdetermined system solve. If we compared the least-square solution returned by USYMQR (termined with flag 7: iteration reached maximum) and LSQR, we found that the solutions are indeed very close (figure 10). These two facts suggest that the problem might be inherent to the algorithm and not the implementation, although further numerical and theoretical investigation is needed to draw a meaningful conclusion.

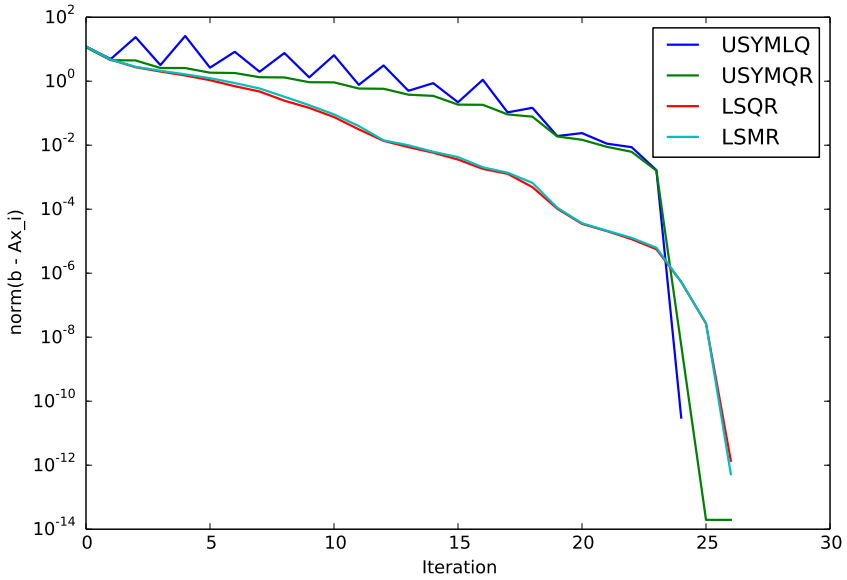

Figure 4: $M = 50$, $N = 25$, A and b are randomly initialized. A is compatible

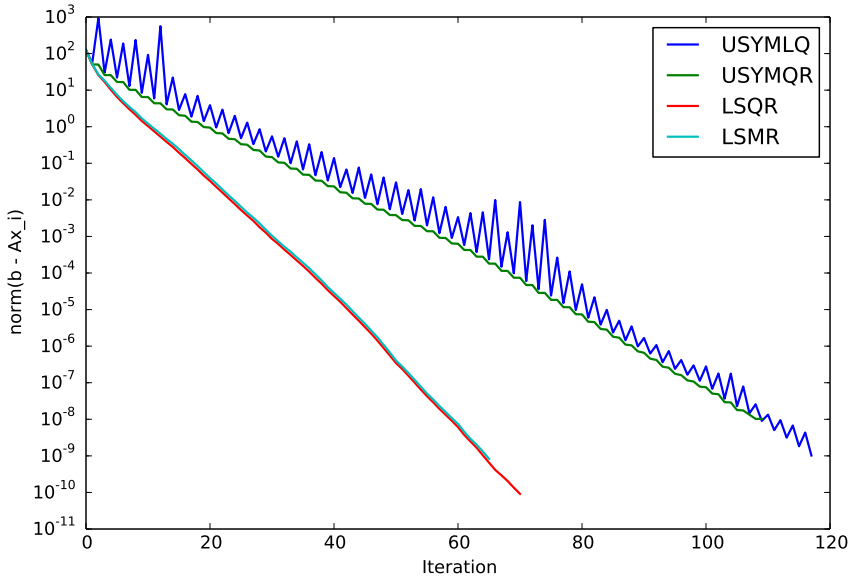

Figure 5: $M = 500$, $N = 250$, A and b are randomly initialized. A is compatible

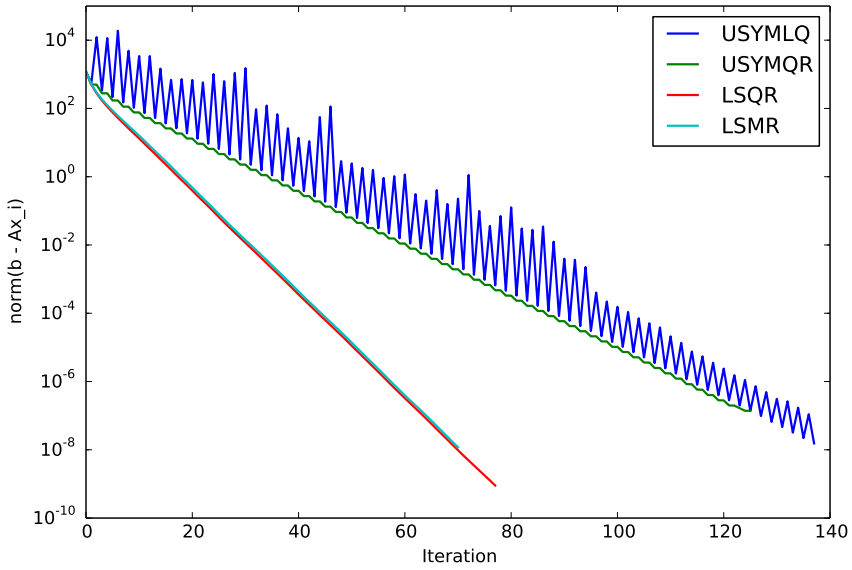

Figure 6: $M = 5000$, $N = 2500$, A and b are randomly initialized. A is compatible

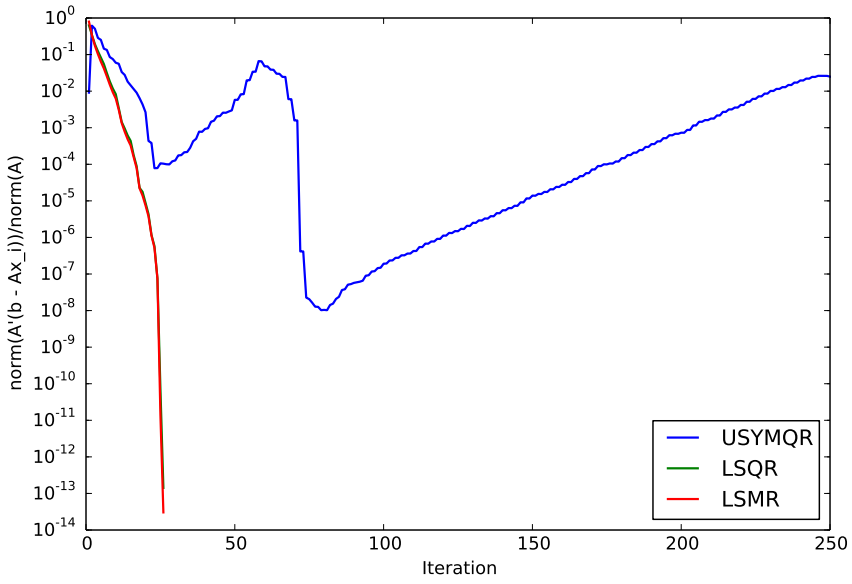

Figure 7: $M = 50$, $N = 25$, A and b are randomly initialized. A is incompatible

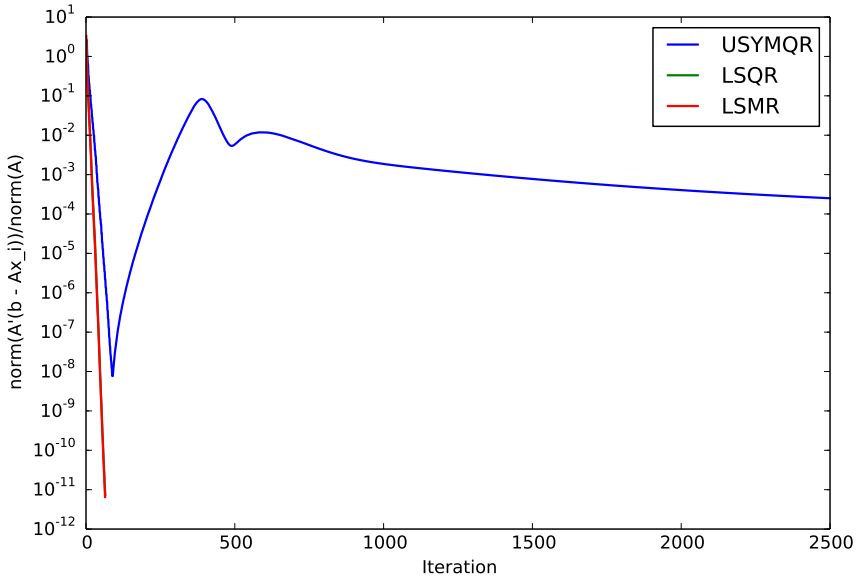

Figure 8: $M = 500$, $N = 250$, A and b are randomly initialized. A is incompatible

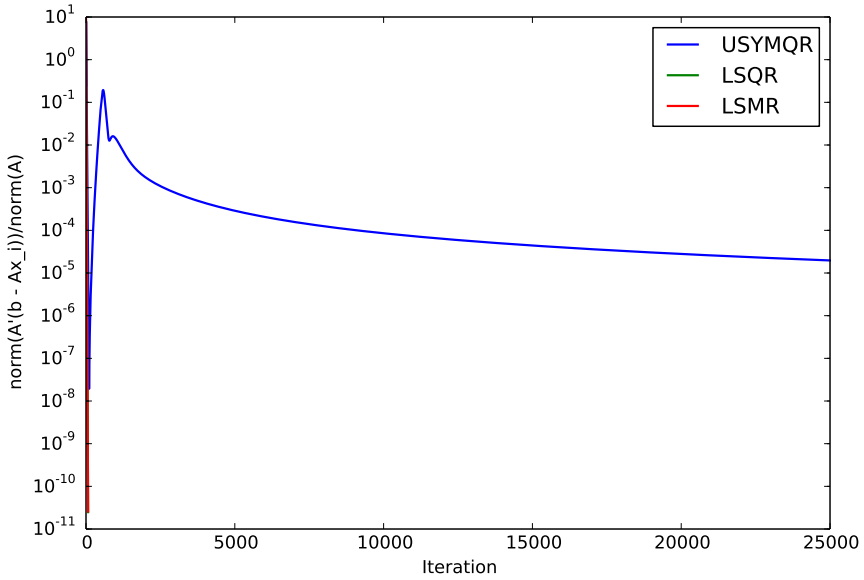

Figure 9: $M = 5000$, $N = 2500$, A and b are randomly initialized. A is incompatible

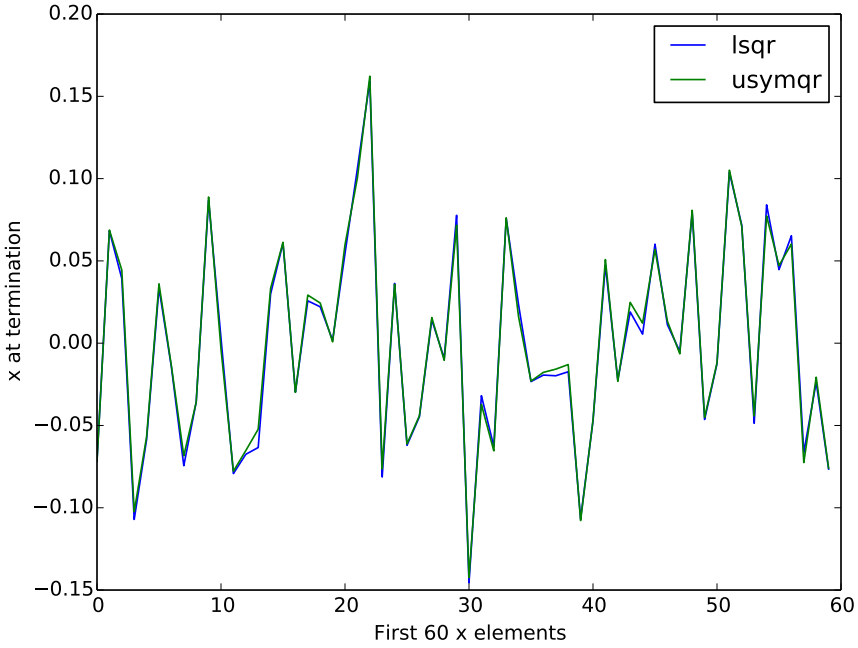

Figure 10: Comparison between the solutions returned by USYMQR and LSQR. The model problem used was the $M = 500$, $N = 250$ case.

5.2 UFL Sparse Matrix Collection

The UFL sparse matrix collection [3] is a large collection of sparse matrices that is open to public. Most of the matrices come from real-world problem, including structural engineering, computational fluid dynamics, computer graphics/vision, robotics, optimization, circuite simulations etc., to name just a few. This impressive collection provides an excellent source of test problems for our USYMLQ and USYMQR algorithms.

The main goal for this testing is to see how many problems can the USYMLQ and USYMQR handle. As of September, 2017, the UFL collection has 1517 matrices that has real entries. We restrict our discussion on only unsymmetric, real matrices but our algorithms and implementation can be readily extended to handle complex and integer matrices (only test modules and proper linear algebra classes need to be rewritten). Out of these matrices, we further restrict ourselves to ones that have less than 10000 rows and 10000 cols, mainly due to constraint on time and computational power.

We found in total 1017 matrices that satisfy the constraints mentioned above. These matrices were downloaded using an in-house Java Script code in the Matrix Market format. I/O class for this format in C++ was written in order to integrate them seamlessly into the pipeline. These matrices have various dimensions and we made no attempt to probe its properties (e.g. its rank, singular values, etc., are all unknown). Please see figure 11 for the distribution of the problem sizes. To speed up the process (solving 2034 problems in total), we multi-threaded the testing program. The whole run can be completed in tens of minutes on a Xeon workstation. All problems were run with the tolerance $a_{tol} = b_{tol} = 10^{-12}$, and maximum iteration $\texttt{maxItn} = 20\max(M, N)$.

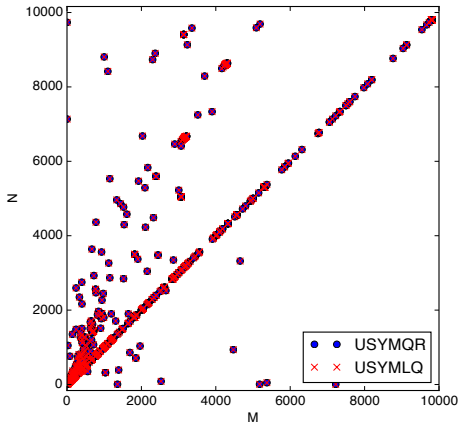

Figure 11: This figure shows the distribution of problem sizes in the test set extracted from the UFL collection. Note that there are actually more under-determined problems than over-determined problems. This bias is in the collection itself rather than our conscious selection. We ran both solvers, USYMLQ and USYMQR, on each of the problem, generated in total 2034 solver results that need processing.

We were able to obtain reasonable solutions for the majority of the problems. However, there are some matrices that give our solver troubles. If residual reduction at 10^{-3} is needed, then USYMLQ solves 79.8% of the problems and USYMQR solves 98.5%; if residual reduction at 10^{-6} is needed, then USYMLQ solves only 43.2% of the problem and USYMQR solves 72.2% of the problems. These statistics are deduced from the data we show in table table 1 and figure 12. A rather large portion of them returned with stop flag 7, which means iteration limits were reached. It is possible that we can improve the residual at the cost of more iterations, but this is not desired as the largest case we have in the test set is 9801×9801, and the iteration limit was set to 0.2 million already. In general, our experiments seem to suggest that USYMQR is more general compared to USYMLQ, and should be tried first when a problem of unknown properties is encountered. In terms of computational speed, USYMQR seems to be slightly faster compared to USYMLQ

on many problems (see figure 13).

Solver Type	Stop flag	Count
USYMLQ	1	400
	7	594
	11	20
USYMQR	1	644
	7	356
	11	14

Table 1: Summary of stop flag.

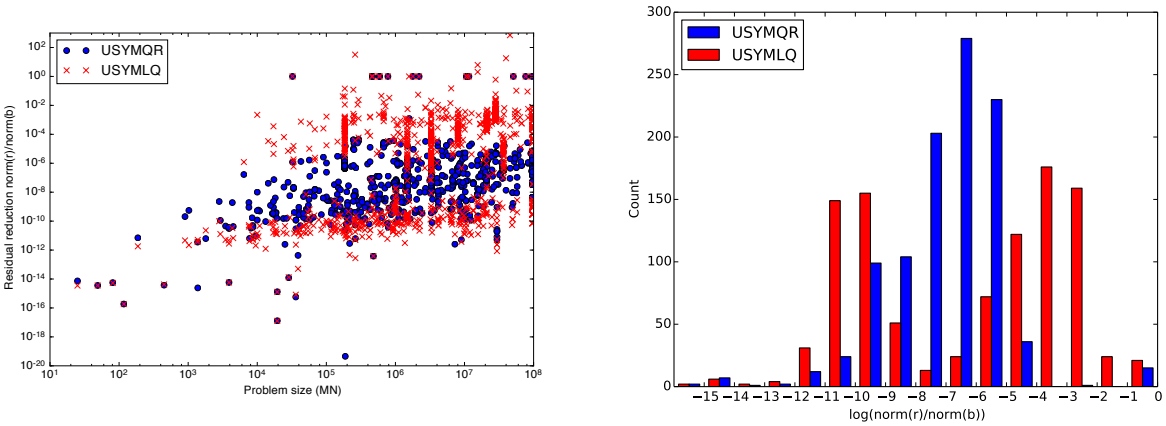

Figure 12: Left: Residual reduction plotted against problem size. Both solvers are run for all problems (blue: USYMQR; red: USYMLQ). Right: Cumulation count of the residual reduction.

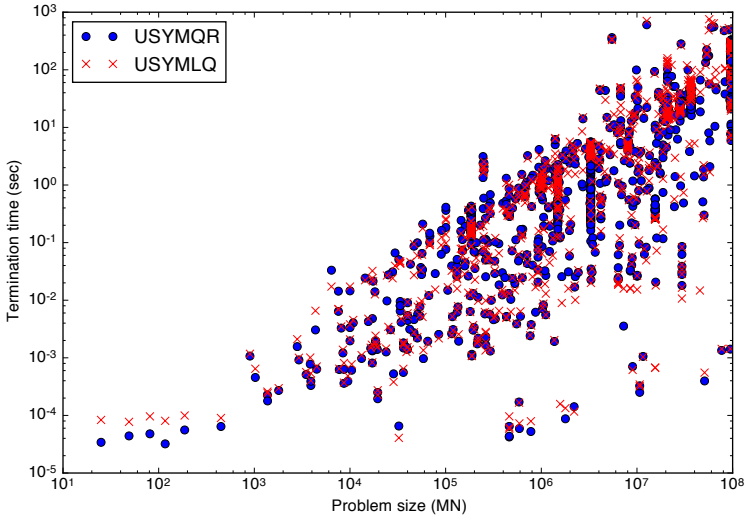

Figure 13: This figure shows the statistics of the timing. We timed the solver from initialization to final termination and return the total elapsed time. This is done on a workstation with Intel Xeon E5-2690V3 2.6GHz 12-core processors.

6 Conclusion

In conclusion, we implemented the USYMLQ and USYMQR in C++. Both algorithms were compared to LSQR, LSMR, and BiCGStab for solving dense random problems. We then constructed a test set based on Tim Davis' UFL sparse matrix collection. About two thousand instances were run to test the robustness of USYMLQ and USYMQR, and it was found that reasonable solutions can be obtained for the majority of them.

References

[1] M. A. Saunders, H. D. Simon, and E. L. Yip. Two conjugate-gradient-type methods for unsymmetric linear equations. *SIAM Journal of Numberical Analysis*, 25(4), 1988.

[2] L. Reichel and Q. Ye. A generalized lsqr algorithm. *Numberical Linear Algebra with Applications*, 15(7), 2008.

[3] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1), 2011.

[4] Eigen. `eigen.tuxfamily.org`, 2017. [Online; accessed 02-September-2017].

[5] Eigen benchmark. `http://eigen.tuxfamily.org/index.php?title=Benchmark`, 2017. [Online; accessed 02-September-2017].

[6] Lsmr: An iterative algorithm for least-squares problems. `https://www.mathworks.com/matlabcentral/fileexchange/27183-lsmr--an-iterative-algorithm-for-least-squares-problems`, 2017. [Online; accessed 03-September-2017].