MS&E 318 (CME 338) Project: USYMQR

Greg DePaul*
Stanford University
Computational and Mathematical Engineering
gdepaul@stanford.edu

Stephanie Sanchez*
Stanford University
Computational and Mathematical Engineering
ssanche2@stanford.edu

June 12, 2018

Abstract

*We have implemented Unsymmetric QR (**USYMQR**) for Julia. We show its performance with various inputs of matrices and benchmark with the following methods: Least Squares QR (**LSQR**) and Biconjugate Gradients Stabilized Method (**BICGSTABL**).*

1 Introduction

1.1 Motivation

Large-sale numerical optimization is present in real-world applications and it is increasingly valuable to be able to solve problems for these applications by preprogrammed algorithms. Many languages have created libraries to cultivate algorithms such as LSQR, MINRES, CG, etc. We have chosen to implement USYMQR to solve

$$\min \|Ax - b\|_2$$

where A is $n\text{x}n$ as well as make the algorithm compliant for the IterativeSolvers library in Julia. We chose Julia because the language is rapidly growing in popularity for its computational power. We also chose to write the algorithm in the same fashion that Julia has written its libraries of functions, purely for the experience of writing framework-backed code.

1.2 Related Work

Previous work begins with Saunder, Simon, and Yip's paper [**?**] that proposes an algorithm for both USYMLQ and USYMQR as well as sometime later producing F90 implementation for USYMLQ. There is also a MatLab implementation of USYMQR provided by Ron Estrin, Phd student at ICME, Stanford University.

1.3 Inputs and Outputs

For our implementation, we wanted to maintain similarity with Julia's function call for MINRES. Therefore, we define the function call:

```
x, hist = usymqr(A, b, maxiter, tol, log)
```

[0]*All authors contributed equally to this work.

Here, we given the least-squares problem $\min_y \|Ay - b\|$, this function outputs the value x as well as a convergence history. The defined functions has several options:

- `maxiter`

  the maximum number of iterations allowed to converge.

- `tol`

  the tolerance for which the res-norm must be within to satisfy the problem constraints. Mathematically:

  $$\|Ay - b\| < \epsilon_{tol}$$

- `log`

  allows for logging within the convergence history object.

In the spirit of programming for the intention of integrating the algorithm within the Julia library, we sought incorporating history search and functionality. Such functionality includes being able to make queries such as:

```
hist.isconverged
hist[:resnorm]
```

which provides insight to users on how well their systems may yield convergence.

2 USYMQR Algorithm

USYMQR effectively utilizes the orthogonal tridiagonalization algorithm for an unsymmetric matrix (see algorithm **??**) and solves the least squares subproblem using QR factorization as described by [**?**] and shown in algorithm **??**.

Algorithm 1 Orthogonal Tridiagonalization Algorithm for an Unsymmetric Matrix

Pick two arbitrary vectors $b \neq 0$, $c \neq 0$
Set $p_0 = q_0 = 0$, $\beta_1 = \|\|b\|\|$, $\gamma_1 = \|\|c\|\|$, and $p_1 = \frac{b}{\beta_1}$, $q_1 = \frac{c}{\gamma_1}$, $maxiters = 10 * n$
For i = 1,2,3,... maxiters
    $u = Aq_i - \gamma_i p_{i-1}$
    $v = A^* p_i - \beta_i q_{i-1}$
    $\alpha_i = p_i^* u$
    $u = u - \alpha_i p_i$
    $v = v - \alpha_i^* q_i$
    $\beta_{i+1} = \|\|u\|\|$
    $\gamma_{i+1} = \|\|v\|\|$
    if $\beta_{i+1} = 0$ or $\gamma_{i+1} = 0$: stop
    else $p_{i+1} = \frac{u}{\beta_{i+1}}$, $q_{i+1} = \frac{v}{\gamma_{i+1}}$

Algorithm 2 USYMQR Algorithm

1: Input: $A \ni A^T \neq A, b, tol = 1e-6, AAnorm = 0, \beta_1 = \||\|b\||\|, x_0 = \overrightarrow{0}, \gamma_{prev} = 0,$
2: $\sigma =, \bar{s} = \gamma, rhs1 = \beta_1, q1 = q2 = 0, w1 = w2 = \overrightarrow{0}, \tau = 0, c_{prev} = c = 1, s = 0, maxiters = 10 * n$
3: For t = 1,2,3,... maxiters do
4: $\quad \alpha, \beta, \gamma, v_{prev} = Tridiagonalization(A)$
5: $\quad \sigma = c * \bar{s} + s * \alpha$
6: $\quad \bar{r} = -s * \bar{s} + c * \alpha$
7: $\quad \tau = s * \gamma$
8: $\quad \bar{s} = c * \gamma$
9: $\quad$ if $t = 1$
10: $\qquad \bar{s} = \alpha$
11: $\qquad \bar{s} = \gamma$
12: $\quad \rho = \sqrt{\bar{r}^2 + \beta^2}$
13: $\quad c = \bar{r}/\rho$
14: $\quad s = \beta/\rho$
15: $\quad rhs1, rhs2 = c * rhs1, -s * rhs1$
16: $\quad w3 = (v_{prev} - \sigma * w2 - \tau_{prev} * w1)/\rho$
17: $\quad x = x + rhs1 * w3$
18: $\quad w1 = w2$
19: $\quad w2 = w3$
20: $\quad r_{norm} = |rhs2|$
21: $\quad q1 = -c_{prev} * s$
22: $\quad q2 = c$
23: $\quad$ if $|r_{norm}| \leq tol \| t \geq maxiters \| condition1 \| condition2$
24: $\qquad$ stop
25: $\quad rhs1 = rhs2$
26: $\quad \gamma_{prev} = \gamma$
27: $\quad c_{prev} = c$

We list the additional stopping conditions as

$$condition1 : r_{norm} * \|[\gamma_{prev} * q1 + \alpha * q2; \gamma * q2]\| / \sqrt{AAnorm} * r_{norm} \leq tol$$

and

$$condition2 : r_{norm} < tol * \sqrt{AAnorm} + tol$$

where $AAnorm = AAnorm + \alpha^2 + \beta^2 + \gamma^2$. We also note that line 4 in algorithm **??** returns the updated values from algorithm **??** for ***one*** iteration of the orthogonal tridiagonalization algorithm.

3 Solver Implementation in Julia

For our solver, we employed the Iterable design pattern. This requires reformatting an algorithm such that it fits within the context of:

```
iterable = usymqr_iterable!(x, A, b, ...)
while !iterable.converged()
        iterable = iterable.next()
end
```

We created a class in Julia, which we called **usymqr_iterable**, that provides methods **next** as well as **converged**. An algorithm that is *Iterable* allows for other developers to later append components of their algorithms to the inner iterations of USYMQR, while maintaining the latest state of USYMQR's run in order to continue to solve the current least squares problem.

Implementing such an Iterable algorithm also allows us to call internal library functions of Julia to populate the convergence history variable, which proved to be very useful in plotting these search histories.

4 Numerical Performance on Usymmetric Matrices

To test our solver, we measured the observed convergence history of a variety of ill-conditioned matrices available from `http://www.math.sjsu.edu/singular/matrices/`. We chose the range of condition values from relatively small condition numbers, to infinity in the case of the NASA / Barth matrix. The matrices we chose also include symmetric and unsymmetric matrices.

We notice that for small matrices, BiCG worked well, but tended to never converge for any matrices with condition numbers larger than $1e13$. On the other hand, Table 1 shows that both USYMQR and LSQR both converge. However, this table may be a little misleading, since the tolerance definitions of USYMQR and LSQR differ in such a way that LSQR may terminate at a smaller Residual than ours.

Table 1: Number of iteration steps required to achieve the requested accuracy

Problem	i_laplace_100	Regtools heat500	Regtools heat200	Regtools heat100	Hollinger / g7jac010	Lucifora / cell1	Nasa / barth
Condition Number	$1e^{50}$	$5.5e^{270}$	$3.6e^{152}$	$7e^{82}$	$6e^{18}$	$1.7e^{12}$	∞
LSQR	31	165	1,000	1,000	1,549	2,446	580
BiCG	10	NC	NC	NC	NC	NC	NC
USYMQR	34	1,582	1,094	551	NC	6,041	45,111

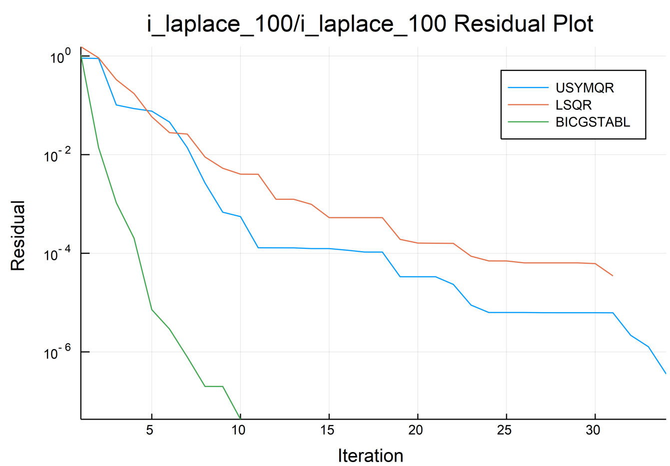

Figure 1: $N = 100, M = 100$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

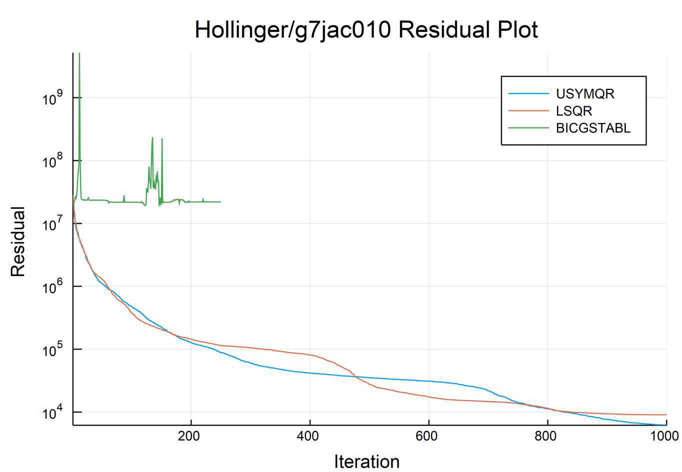

Figure 2: $N = 2880, M = 2880$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

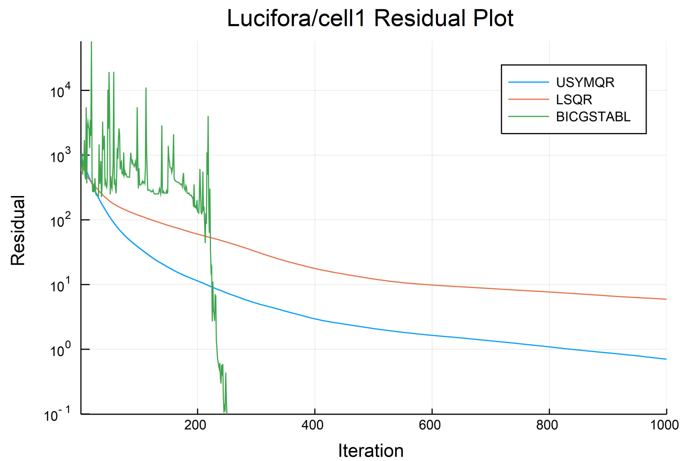

Figure 3: $N = 7055, M = 7055$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

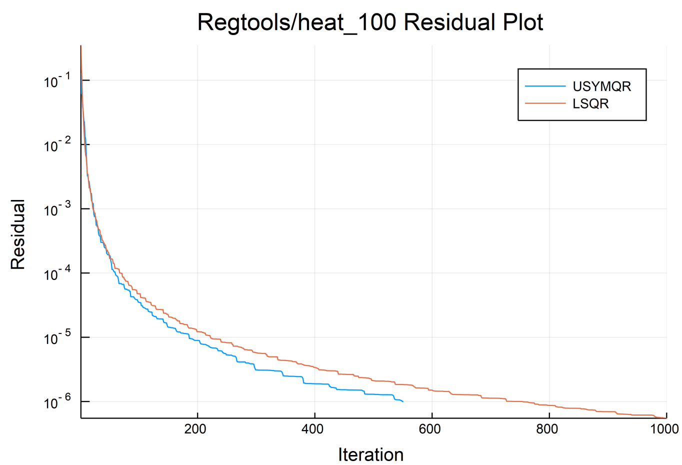

Figure 4: $N = 100, M = 100$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

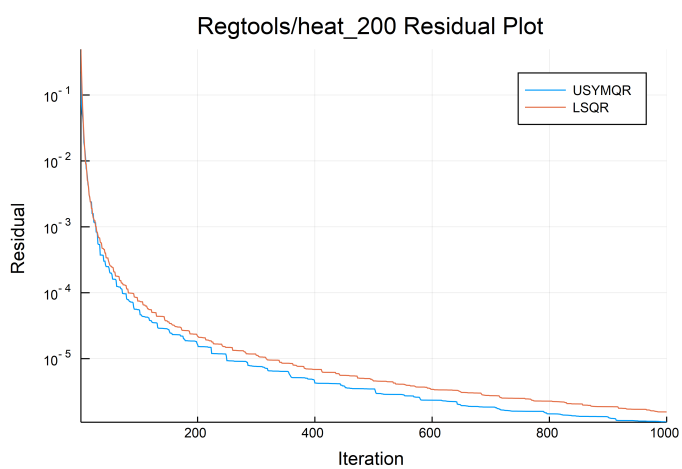

Figure 5: $N = 200, M = 200$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

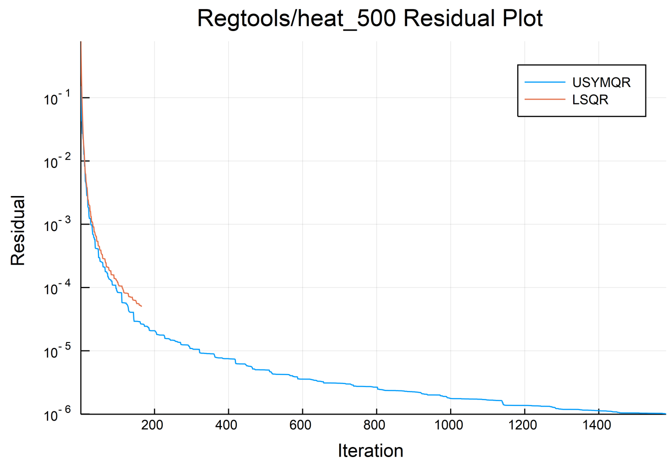

Figure 6: $N = 500, M = 500$, and $b = A\mathbb{1}$ where x is initialized to the zero vector.

For the most part, we see that USMQR and LSQR tend to perform very similarly. For Lucifora, we see immediately that USYMQR performs a lot quicker than LSQR. It could be that LSQR uses more resources as opposed to USYMQR for relatively-good conditioned matrices. As the condition numbers increases, the algorithms tend to perform identically.

Admittedly, we should have tested on rectangular matrices, but we're confidant that USYMQR will perform comparably to its performance on square matrices.

5 Future Work

The algorithm we constructed only performs over real, floating-point

`{Float32, Float64 }`

matrices in the Julia language. However, the iterative solvers integrated in the Julia language include support for

`{Complex64, Complex128 }`

So if there is desire to integrate USYMQR into the Julia language library, it may be necessary to extend this support.

6 Acknowledgements

This project is done for CME 338: Large-Scale Numerical Optimization at Stanford University. Many thanks to Michael Saunders for his help and guidance throughout this project. You can access the project at

`https://github.com/gregdepaul/USYMQR`

References

[1] M. A. Saunders, H. D. Simon, E. L. Yip *The Conjugate-Gradient-Type Methods For Unsymmmetric Linear Equations*.

[2] Lothar Reichel, Qiang Ye *A generalized LSQR algorithm*.