Investigation of the Accuracy and Performance of Condition Number Estimation of a Square Matrix

Haocheng An
Institute for Computational Engineering and Sciences
Department of Computer Science
Department of Mathematics
The University of Texas at Austin
Email: haochengan@utexas.edu

Yiran Shen
Department of Mathematics
Department of Finance
The University of Texas at Austin
Email: shenyiran91@utexas.edu

Supervisor: Robert A. van de Geijn

August 12, 2016

Abstract

The condition number of a matrix quantifies how accurately the solution to $Ax = b$ is computed. More specifically, it equals the maximum by which a relative error in the right-hand side b can be magnified into the relative accuracy of the solution. In this study we examine Hagers method for estimating the condition number of a matrix when using the one-norm and infinity norm and see if any performance can be enhanced by other methods. A preliminary result shows that while LAPACK generally works well, the performance can be enhanced in some cases while still maintains reasonable results.

1 Introduction

LAPACK is one of the most popular numerical linear algebra software package. It offers many useful implementations of the matrices operations with understandable code. Besides providing reasonable results for users, LAPACK also gives us some auxiliary routines to better understand the result. DGECON, One of these auxiliaries, is used to give feedbacks on how hard the problem is in terms of small relative errors in input can magnify the output error. This is also known as "*condition number*" problem.

BLIS, BLAS-like Library Instantiation Software Framework [6], is another numerical linear algebra library but concentrate on the high performance of the code. This library is originally implemented in C with base level operation "primitives".

So we first implement the condition number estimation with the basic "primitives". Later, we change some features of the matrix to see if the accuracy and performance will have any difference.

2 Background

In order to derive the formula for computing the condition number of a matrix, we need to provide following mathematical background first.

2.1 Norms

2.1.1 Vector Norms

Vectors are segments with a specific directions and sizes. The key components of a vector is the direction and the size. The direction can be quantified by unit vector. As to the size, in other words, how big or how small a vector is, we use *vector norms* to quantify it.

The $p - norm$ of a vector x is defined as

$$||x||_p = \begin{cases} (\sum_{i=1}^{n} |x_i|^p)^{\frac{1}{p}} & \text{if } 1 \leq p < \infty \\ max|x_i| & \text{if } p = \infty \end{cases}$$

2.1.2 Matrix Norms

A matrix can be regarded as a functor applied to vectors, which causes linear transformations on vectors. Sometimes people are interested in how much effect can a matrix have at most on a vector norms, so corresponding to vector norms, we define *matrix norms*.

For a given matrix A_{m*n}, and any vector x of dimension n, we define the $p-norm$ of matrix A as follows

$$||A||_p = \max_{x \neq 0} \frac{||Ax||_p}{||x||_p}$$

2.2 Theorems

With the definitions above, we can derive following theorems.These are our math "primitives" for future algorithm understanding.

By convention, from this point on, we use upper case letters to denote matrices, lower case letters to denote vectors and greek letters to denote constants.

$1 ||Ax|| \leq ||A|| * ||x||$

Proof: According to the definition of matrix norm, $||A||_p \geq \frac{||Ax||_p}{||x||_p}$. When we put $|x||$ to the same side as $||A||$, we have the above theorem.

$2 ||A||_p = \max_{||x||_p=1} ||Ax||_p$

Proof: $||A||_p = max \frac{||Ax||_p}{||x||_p} = max \frac{||\kappa A\hat{x}||_p}{||\kappa \hat{x}||_p} = max||Ax||_p$

κ is the constant such that $x = \kappa\hat{x}$, where $\hat{x}$ is the unit vector with same direction as x.

$3\ ||A||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |A_{ij}|$

Proof $||A||_1 = \max_{||x||_p=1} ||Ax||_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |A_{ij}|$

$4\ ||A||_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |A_{ij}|$

Proof:$||A||_\infty = \max_{||x||_p=1} ||Ax||_\infty = \max_{1 \leq i \leq n} \sum_{j=1}^{n} |A_{ij}|$

$5\ ||A||_1 = ||A^T||_\infty$

Proof: This can be directly derived from Theorem 3 and Theorem 4.

2.3 Conditioning

We now want to explore how the perturbed of b can have effects on the result of x [4]

For the exact equation, we have

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

As b is perturbed, Actually we are solving

$$A(x + \delta x) = b + \delta b \tag{2}$$

We want to explore an expression for the following κ

$$\frac{||\delta x||}{||x||} \leq \kappa \frac{||\delta b||}{||b||} \tag{3}$$

According to equation(1) and Theorem 1,

$$||b|| \leq ||A|| * ||x|| \tag{4}$$

Similarly, When we subtract (1) from (2) and apply theorem 1 to the original equation we have

$$||\delta x|| \leq ||A^{-1}|| * ||\delta b|| \tag{5}$$

Rewrite equation (4) to fit the equation pattern in (3), we get

$$\frac{1}{||x||} \leq \frac{||A||}{||b||} \tag{6}$$

Multiply (5)(6) together,

$$\frac{||\delta x||}{||x||} \leq ||A||||A^{-1}|| \frac{||\delta b||}{||b||} \tag{7}$$

The condition number is denoted and defined as $\kappa(A) = ||A|| * ||A^{-1}||$

3 Algorithms

From the definition of last part, the upper bound for this uncertainty only determined by the matrix. Now the problem comes to how to find the value of two multipliers. As we only concentrate on the 1-norm and infinity-norm of a matrix given matrix A, $||A||$ can be computed by iterates through the column vector or row vectors respectively and maximize the one norm of it. The key part lies on the computation of $||A^{-1}||$. Direct inverse the matrix and then find vector norm is an $O(n^3)$ operation. We examine the code for LAPACK. It gives us Hager's algorithm, which is based on convexity theory and gradients. [2] As this paper mainly focus on the performance of the code, we will not go deep in the algorithm.

3.1 Hager's Algorithm

The algorithm for computing the $||A^{-1}||_1$ norm of a matrix is (The expression in the last parenthesis(if any) denotes number of FLOPs(the cost of computation) required for that specific steps, which will be discussed later on.) [2, 5]

(1) Choose $x = n^{-1}(1, 1, 1 \cdots, 1)^T$

(2) Factorize $A = LU$ $(\frac{2}{3}n^3)$

(3) Solve $Lb = x$ and $Uy = b$, (Solve $Ay = x$)$(2n^2)$

(4) Set $v = sign(A^{-1}x) = sign(y)(n)$

(5) Solve $U^T d = v$ and $L^T g = d$(Solve $A^T g = v$)$(2n^2)$
(6) If $||g||_\infty \leq g^T x$,$(\frac{n}{2} + 2n)$
$est||A^{-1}||_1 = ||A^{-1}x||_1$
Otherwise $x = e_j$ where $|g_j| = ||g||_\infty$ and return to step(3)
(7) Get pass the loop guard

When we are dealing with l_∞ norm, given theorem 5 and the property of the transpose of a matrix

$$A = LU \Rightarrow A^T = U^T L^T$$

We can use the similar algorithm with above one but substitute several matrices in Step 3 and Step 5 namely,

$$L \to U^T$$

$$U \to L^T$$

$$L^T \to U$$

$$U^T \to L$$

After these substitutions, we can find $||A^{-1}||_\infty$.

For the norm of triangular matrices, the LU factorization is unnecessary. Also, $A = LU$ and either L or U can be the identity matrix, so two of the above four triangular solve routines are not required.

3.2 Cost

Now we discuss the contents inside the parenthesis in the last subsection.

We first define flops. *Flops* is the number of operations need to compute in a unit time. In previous section, we use n to denote the dimensions of a matrix. The contents in the parenthesis is the specific flops required of that step. As we deal with matrices with thousands of entries, if two steps are of different order on n, the one with bigger exponents is harder to compute. If they are of same order, the one with bigger coefficients is harder. For a step or routine, usually the step with which is hardest to compute determine the general hardness of the step or routine.

According to Theorem 5 and illustration in the previous subsection, the flops required are same when computing the l_1 and l_∞ norm. Usually it takes about 2 iterations to stop. So we times the number of flops in (3)-(6) by 2 and sum it up, the flops routine is of order $O(n^3)$

For triangular case, as LU-factorization and half of the triangular solves operations are not necessary, most of the work lies on the triangular solve. So it is in general an order $O(n^2)$ algorithm.

3.3 The LAPACK Routine

However, when carefully examine DGECON, the LAPACK routine name for condition number estimation, it requires the norm of A as a parameter to pass in. [3]

```
SUBROUTINE DGECON( NORM, N, A, LDA, ANORM,
 RCOND, WORK, IWORK, INFO )
```

In the comment part, the routine requires the matrix to be its LU factorized form. If not familiar with the routine, users could input a wrong matrix to get wrong result.[3]

```
A       (input) DOUBLE PRECISION array, dimension (LDA,N)
        The factors L and U from the factorization
        A = P*L*U      as computed by DGETRF.
```

Also, when we look carefully in the DGECON routine, GOTO coding styles are applied, which is considered harmful.[1] One of the reason lies on the hardness of tractable of the code with GOTO method. Users have to jump around the code several times to get the flow of the code. [3]

```
IF( SCALE.LT.ABS( WORK( IX ) )*SMLNUM
.OR. SCALE.EQ.ZERO )
$       GO TO 20
            CALL DRSCL( N, SCALE, WORK, 1 )
               END IF
        GO TO 10
END IF
```

4 Performance and Accuracy

4.1 Generation of Testing Matrices

With the drawbacks in LAPACK mentioned above, we now implement our own condition number estimation routine with same algorithms but based on different libraries

For the implementation using BLIS library, we use the reference guide[6, 7].

For both performance and accuracy of the routine, the matrices are stored by column major. Also, we use matrices whose entries are random numbers generated from C libraries from interval $(0, 1]$. To achieve this, we use the following code

```
(double) rand() / (double) RAND_MAX
```

For the triangular case, we use lower triangular matrix. According to the symmetrical property, upper matrices should give us similar result. We set the diagonal entries to be 1 This is because if we still use random number, the expect

value of each diagonal entries to be 0.5. When we deal with big matrices, for example 1000 entries, the determinant would be

$$(\frac{1}{2})^{1000} = \frac{1}{(2^{10})^{100}} < \frac{1}{(10^3)^1 00} = \frac{1}{10^{300}}$$

The matrix is nearly singular so that the condition number cannot work well.

Now let us discuss the size of those candidates matrices. For full matrices, we set the upper bound to be 10000. For triangular matrices, we set it to be 5000. Based on the observation of the condition number, for the triangular case, usually it grows dramatically with the growth of the size of the matrix. When size reaches about 5000, condition number is about 10^{100}. As the computer is using 64 bits to represent real doubles, bigger condition numbers may cause overflow and thus we don't go to larger matrices.

4.2 Accuracy graphs

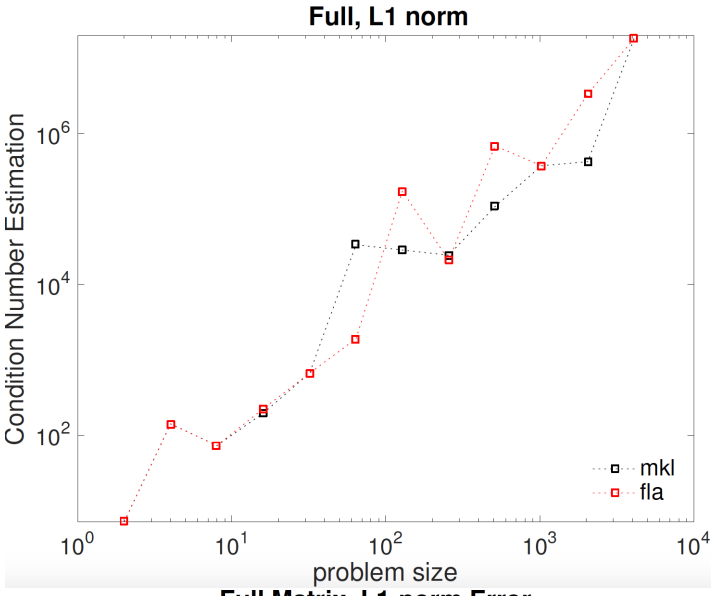
For the accuracy of the implementation, we examine the ratio between the answer we get and the answer given by the mkl implementation.

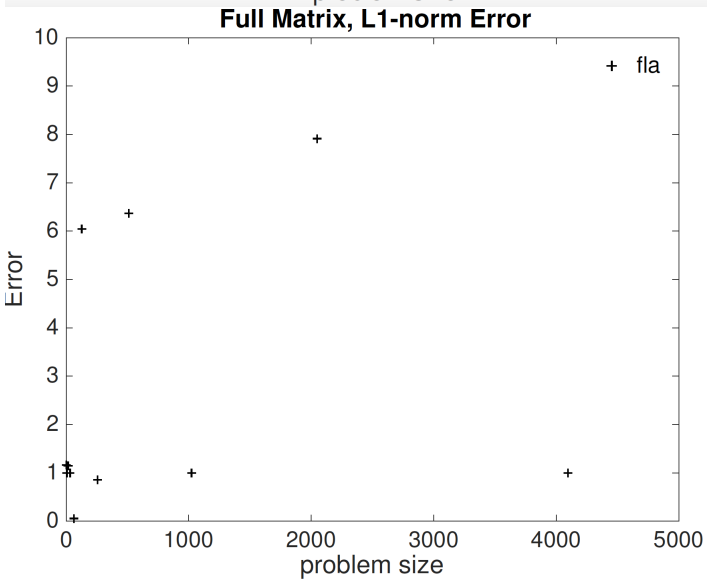
The reason we use the ratio rather than the relative error is determined by the characteristic of the condition number. We only care if the estimation number is of the same order of the theoretical value. If yes, then the estimation is satisfiable. As the same order should be $[0.1, 10]$ times of the actual number, we have the following definition:

An estimation is $satisfiable$ if and only if

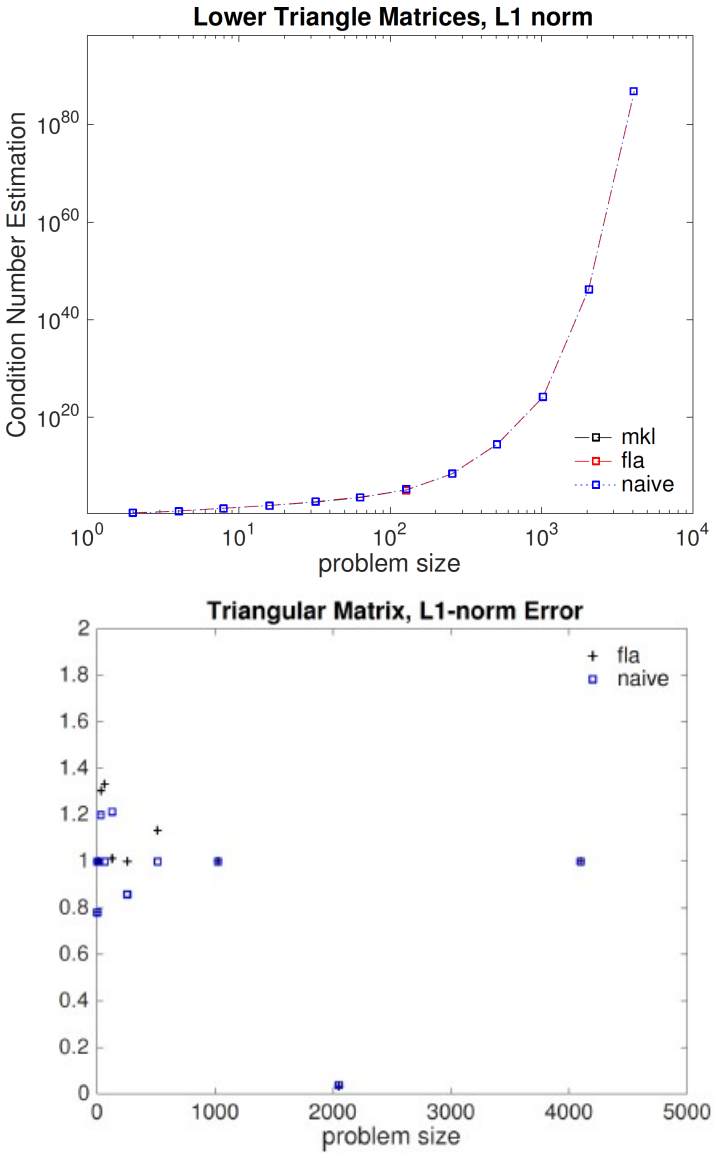
$$\mu = \frac{ActualValue}{TheoreticalValue} \in [0.1, 10]$$

If we use the relative error instead, however, we don't have a corresponding interval. For example, if the relative error is 0.99, we don't know if the ratio $\mu = 0.01$ or $\mu = 1.99$. As $0.01 \notin [0.1, 10]$ but $1.99 \in [0.1, 10]$, If the former one, the estimation is not satisfiable, If the latter one, it is satisfiable.





This pair of graphs shows the estimation and exact condition number of a full matrix A_{n*n} under l_1 norm. The value of n is shown in $x-axis$ of both graphs. First graph show the real value of the estimation and theoretical value of the condition number on $y-axis$. Second graph show their ratios on $y-axis$.



This pair of graphs shows the estimation and exact condition number of a full matrix A_{n*n} under l_∞ norm. The value of n is shown in $x-axis$ of both graphs. First graph show the real value of the estimation and theoretical value of the condition number on $y-axis$. Second graph show their ratios on $y-axis$.

As shown in the graphs, in the full matrix case, almost all y-values of those points are within the satisfiable range.

For the triangular case, estimation works even better. Most of those points are lies within $[0.8, 1.4]$. Even with inaccuracies, only 1 point hits the unsatisfiable range.

4.3 Performance graphs

Now let us consider the performance of these implementations. We use time to reflect such property. In most scientific computing report, number of flops per unit time seems like a reasonable choice. However, the unit flops is hard to tell for the following 2 reasons.

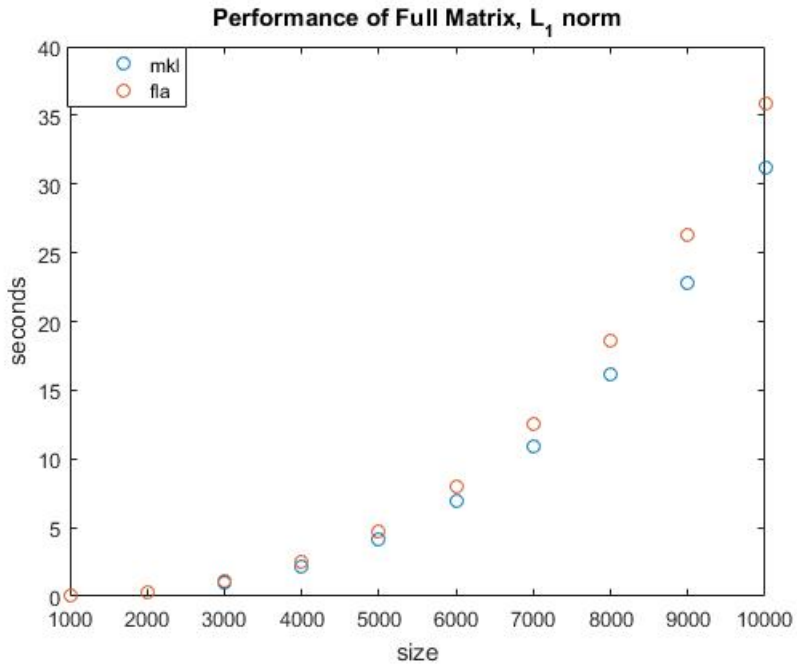
1. In LAPACK routine, involves in too much error checking so the FLOPs is hard to tell.

2. As illustrated in the algorithms part, The loop will end if and only if $||g||\infty \leq g^T x$ Though it usually reaches end in 2 3 iterations, it is still an undetermined and we still need to judge it case by case.

By using time needs to compute, however, we can tell which one is efficient simply by picking out the one using the least.

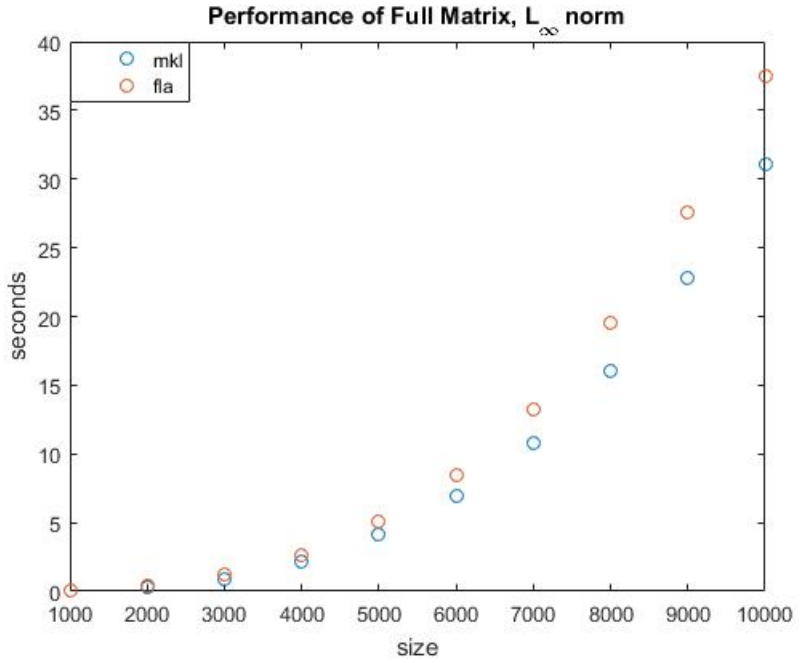
Now let us examine performance graphs.

All data coming from TACC Stampede System (Intel® Xeon® CPU E5-2680, Sandy Bridge, 23.76 GFLOPS peak for single-core, 21.6 GFLOPS/core peak for multi-core) processor using Intel C compiler version 15.0.2 with optimization flag -O2.



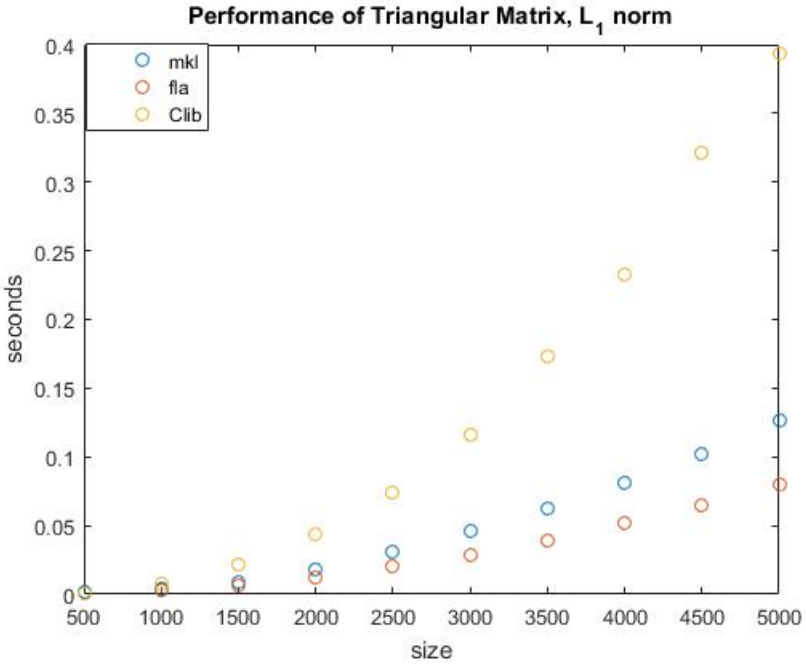
The graphs above shows the performance of the condition number estimation of a full matrix under l_1 norm. The $x - axis$ reflects the n of the matrix A_{n*n} and $y - axis$ reflects the time needs to finish the estimation routine with second unit given by LAPACK and BLIS, which are represented by mkl and fla in the graph respectively. We can see the performance in general:

$$mkl > fla$$



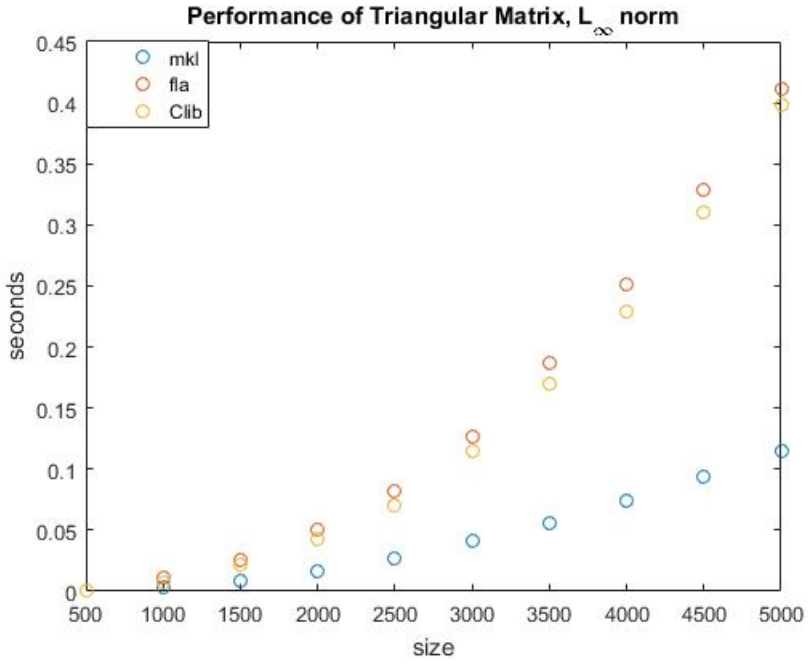
The graphs above shows the performance of the condition number estimation of a full matrix under l_∞ norm. The $x - axis$ reflects the n of the matrix A_{n*n} and $y - axis$ reflects the time needs to finish the estimation routine with second unit given by LAPACK and BLIS, which are represented by mkl and fla in the graph respectively.We can see the performance in general:

$$mkl > fla$$



The graphs above shows the performance of the condition number estimation of a triangular matrix under l_1 norm. The $x - axis$ reflects the n of the matrix A_{n*n} and $y - axis$ reflects the time needs to finish the estimation routine with second unit given by LAPACK, BLIS and C library. which are represented by mkl, fla and Clib in the graph respectively. We can see the performance in general:

$$fla > mkl > Clib$$



The graphs above shows the performance of the condition number estimation of a triangular matrix under l_1 norm. The $x - axis$ reflects the n of the matrix A_{n*n} and $y - axis$ reflects the time needs to finish the estimation routine with second unit given by LAPACK, BLIS and C library. which are represented by mkl, fla and Clib in the graph respectively. We can see the performance in general:

$$mkl > Clib > fla$$

For full matrices , we can see that our implementation is a little slower than the mkl one.

In triangular case, we also implement with C standard library without using any routines coming from BLIS libraries or flame libraries. As it is directly from the C library,we call it Clib. When calculate 1-norm, the fla performs best. For the infinity norm, however, mkl works best.

Also, when we strictly examine the data for these three groups of numbers, smaller matrices any use of the library won't make much difference. Also, we notice the significance in the difference of the performance of fla implementation in L_1 and L_∞ norm. This may coming from the fact that all testing matrices are column major stored. When compute $||A||_\infty$, the compiler have to iterates through rows and the distance between two adjacent entries in a row equals the size of the matrix. Such accessing did not use the stride feature of the caches and thus negatively effect the performance. Similar issues also occur when performing the triangular solve subroutine.

When we compare the graph with full matrix performance and lower triangular matrices performance, we can see that the blis library performs better in the latter one but worse in the first one.

As the number of flops required illustrated in the "ALGORITHMS" part, the most time-consuming in the full matrices is LU factorization while triangular solve in the triangular matrices case. So we now know that the hardness of the performance in this routine is LU factorization.

5 Conclusion

Condition number is a key character of quantifying the errors in a linear system of equations $Ax = b$. It provides an upper bound for the effect of relative error of b can have on x. It is defined as $\kappa(A) = ||A|| * ||A^{-1}||$. However, directly inverse the matrix is an expensive operation. Even if accuracy works, other issues like round -off errors will make the algorithm unreliable. When we look carefully into LAPACK, it shows the calculation of the inverse norm under Hagers algorithm without actually invert the matrix.

However, given some drawbacks in the routine, we implement it in $BLIS$ library. It does not work any better for the full matrix but have a better performance for the computing l_1 norm.

Also, we find the libraries works relatively better than C implementation in those big matrices with dimensions over 1000. For the matrix that dimension less than 1000, C library implementation works almost as same as the high performance library.

Future work would probably improve the LU factorization in the flame library. Also, write a better LU factorization algorithm for Clib implementation. We do have the LU factorization but it is rather inaccurate for matrices with dimensions over 50. So we did not include the result of that in the Performance and Accuracy part.

Further, we can include the error checking, upper bound and lower bound checking of entries of the matrix.

Acknowledgment

The authors would like to thank Robert A van de Geijn for his patient instruction and supervising. Also would like to thankful to Devin Matthew, Jianyu Huang, Tyler M. Smith, Field G. Van Zee in the Science of High-Performance Computing group for their generous help.

References

[1] Dijkstra, E, A case against the Goto statement, Communications of the ACM, volume 11, pg 147, 1968.

[2] Philip Lance Shuler A search for Lapack condition estimator Counterexamples Dec 1992.

[3] Lapack Condition Number Estimation
`http://www.netlib.org/lapack/double/dgecon.f`.

[4] Robert A van de Geijn Linear Algebra: Foundations to Frontiers A Collection of Notes on Numerical Linear Algebra
Jan 2016

[5] Higham, Nicholas J. "A survey of condition number estimation for triangular matrices." SIAM(Society for Industrial and Applied Mathematics) Review 29.4 (1987): 575-596.

[6] BLIS API Quick Reference
`https://github.com/flame/blis/wiki/BLISAPIQuickReference.`

[7] Field G. Van Zee libflame The Complete Reference (version 5.1.0-21)
`http://www.cs.utexas.edu/ flame/web/libflame.pdf`