DS master programme at UCU
Linear Algebra 2020-2021

Final report: Solving linear least square problems using LAPACK

Teodor Romanus

February 15, 2021

1 Introduction

The main topic of this project is solving linear least square problems (LLS) numerically using LAPACK.

LAPACK ("**L**inear **A**lgebra **Pack**age") is a software library for solving numerical linear algebra tasks. It provides subroutines [1] to solve linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. Multiple matrix factorization methods are provided, too. [8]

1.1 Motivation

The author has much interest in the area of fundamental math algorithms implementations and low-level optimizations.

LAPACK is widely used in the scientific world, particularly as the base of MATLAB, the `scipy` package for Python. LAPACK API is used in the parallel and distributed libraries such as ScaLAPACK, OpenBLAS, Intel MKL, Arm Performance Libraries, and others.

1.2 Problem

The linear least square problem has the next form:

$$\underset{\mathbf{x}}{\textbf{minimize}} \quad \|A\mathbf{x} - \mathbf{b}\| \tag{1.1}$$

where A is an m-by-n matrix, $\mathbf{b}$ is a given m element vector and $\mathbf{x}$ is the n element solution vector.

[1] *Subroutine* is another name for a function (procedure) in FORTRAN.

2 Algorithms overview

LAPACK has the next subroutines to solve linear least square problems:

1. **xGELS**: solves LLS using QR or LQ factorization;
2. **xGELSY**: solves LLS using complete orthogonal factorization;
3. **xGELSS**: solves LLS using SVD;
4. **xGELSD**: solves LLS using divide-and-conquer SVD.

2.1 LAPACK naming scheme

LAPACK provides the same range of functionality for **real** and **complex** data. All routines are provided in both **single** and **double** precision versions.

All public interface routines have 6-letter names of the form **XYYZZZ**.

The first letter, **X**, indicates the data type as follows:

S REAL
D DOUBLE PRECISION
C COMPLEX
Z DOUBLE COMPLEX

For example, the function that solves the LLS problem using divide-and-conquer SVD with double floating accuracy has name `DGELSD`.

The next two letters, **YY**, indicate the type of matrix. Some algorithms are optimized for specific matrix properties. Some of the most general matrix types:

GE general (i.e., unsymmetric, in some cases rectangular)
HE (complex) Hermitian
OR (real) orthogonal
SY symmetric

`DGELSS` from the example above corresponds to a general matrix type, which means we don't impose any matrix properties while solving LLS.

The last three letters **ZZZ** indicate the computation performed. For example, `SGEBRD` is a single precision routine that performs a bidiagonal reduction (BRD) of a real general matrix.

2.2 xGELS

xGELS is a simple driver function[2]. It solves the problem (1.1) on the assumption that A has a full rank: $\text{rank}(A) = \min(m, n)$.

The decomposition is based on the use of elementary Householder matrices $H = I - \tau \mathbf{v}\mathbf{v}^T$ (the `xGEQRF` function in `LAPACK`).

1. For an overdetermined ($m \geq n$) "tall" matrix a least-square solution is found using QR decomposition:

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix} = Q_1 R, \tag{2.1}$$

[2] *Driver routines* solve a complete problem, for example solving a system of linear equations, or computing the eigenvalues of a real symmetric matrix.

where R is an n-by-n upper triangular matrix and Q is an m-by-m orthogonal (or unitary) matrix, Q_1 consists of the first n columns of Q, and Q_2 the remaining $m-n$ columns.

The solution can be computed as

$$\mathbf{x} = R^{-1}Q_1^T\mathbf{b},$$

where $R^{-1}Q_1^T$ can be computed by a simple back-substitution algorithm, starting with Q_1^T on the right side.

2. For an underdetermined ($m < n$) "wide" matrix a minimum norm solution is found:

$$\begin{aligned} \textbf{minimize} \quad & \|\mathbf{x}\| \\ \textbf{given} \quad & A\mathbf{x} = \mathbf{b} \end{aligned} \tag{2.2}$$

To solve this minimization problem, the LQ factorization is used:

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = LQ_1,$$

where L is m-by-m lower triangular, Q is n-by-n orthogonal (or unitary), Q_1 consists of the first m rows of Q, and Q_2 the remaining $n-m$ rows.

The solution is given by:

$$\mathbf{x} = Q_1^T \begin{pmatrix} L^{-1}\mathbf{b} \\ 0 \end{pmatrix}$$

2.3 xGELSY

This function, as well as the next ones, solves the problem (1.1), allowing for the possibility that A is rank-deficient. In case of the rank-deficient matrices, the LLS problem transforms into the minimum norm problem (2.2).

xGELSY uses a complete orthogonal factorization to solve the LLS problem.

The routine first computes a QR factorization with column pivoting:

$$AP = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \quad m \geq n, \tag{2.3}$$

where Q and R are as before in (2.1) and P is a permutation matrix, chosen (in general) so that

$$|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|$$

and moreover, for each k,

$$|r_{kk}| \geq \|R_{k:j,j}\| \quad \text{for } j = k+1, \ldots, n$$

The equation (2.3) can be rewritten as

$$AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$

with R_{11} defined as the largest leading submatrix with the order of the effective rank of A. Then, R_{22} is considered to be negligible, and R_{12} is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

$$AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z^T$$

The minimum norm solution is then

$$\mathbf{x} = PZ \begin{pmatrix} T_{11}^{-1} Q_1^T \mathbf{b} \\ 0 \end{pmatrix},$$

where Q_1 consists of the first rank A columns of Q.

The matrix Z is not formed explicitly, but is represented as a product of elementary reflectors, also known as elementary Householder matrices:

$$Z = H_1 H_2 \dots H_k, \quad H = I - \tau \mathbf{v}\mathbf{v}^T$$

2.4 xGELSS

xGELSS computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A:

$$A = U\Sigma V^T, \quad (A = U\Sigma V^H \quad \text{in the complex case}),$$

where U and V are orthogonal (unitary) and Σ is an m-by-n diagonal matrix with real diagonal elements, σ_i, such that

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$$

The singular values and singular vectors satisfy:

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^T u_i = \sigma_i v_i \quad (\text{or} \quad A^H u_i = \sigma_i v_i \quad)$$

where u_i and v_i are the i-th columns of U and V respectively.

The solution for a full-rank matrix A:

$$\mathbf{x} = V\Sigma^{-1} U^T \mathbf{b}$$

If the matrix has non-full rank $A = r$, the minimum norm solution is

$$\mathbf{x} = \sum_{i=1}^{r} \frac{\mathbf{u}_i^T \mathbf{b}}{\sigma_i} \mathbf{v}_i$$

2.5 xGELSD

xGELSD computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.

The problem is solved in three steps as described in [1]:

1. Reduce the coefficient matrix A to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).

   Golub and Kahan [5] revealed that any matrix A can be decomposed as

   $$A = UBV^T,$$

   where B is an upper (or lower) bidiagonal matrix and U and V are orthogonal matrices. Moreover, U and V are formed as products of two sequences of Householder transformations. Here, Q_k is chosen to zero the last $m-k$ elements in the k-th column of (b, A) and P_k is chosen to zero the last $n-k$ elements in the k-th row of A. The final result is the decomposition:

   $$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} B \\ 0 \end{pmatrix} V^T, \tag{2.4}$$

   where $\begin{pmatrix} U_1 & U_2 \end{pmatrix} \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices, U_1 is the first n columns of U and U_2 is the remaining $m-n$ columns of U, and

   $$B = \begin{pmatrix} \alpha_1 & \beta_1 & & \\ & \alpha_2 & \ddots & \\ & & \ddots & \beta_{n-1} \\ & & & \alpha_n \end{pmatrix}$$

   is an upper bidiagonal matrix. The matrices U and V are computed as

   $$U = Q_1 Q_2 \dots Q_{n+1} \in \mathbb{R}^{m \times n}, \quad V = P_1 P_2 \dots P_{n-1} \in \mathbb{R}^{n \times n}$$

2. Solve the BLS problem using a divide and conquer approach.

   Bidiagonal divide-and-conquer algorithm recursively divides B into two subproblems [7]:

   $$B = \begin{pmatrix} B_1 & 0 \\ \alpha_k e_k & \beta_k e_1 \\ 0 & B_2 \end{pmatrix}, \tag{2.5}$$

   where $B_1 \in \mathbb{R}^{(k-1) \times k}$ and $B_2 \in \mathbb{R}^{(n-k) \times (n-k)}$ are upper bidiagonal matrices, and e_j is the j-th unit vector-row of appropriate dimension. For example, we might take $k = \lfloor \frac{n}{2} \rfloor$.

   Assume we have the SVDs of B_1 and B_2:

   $$B_1 = Q_1 \begin{pmatrix} D_1 & 0 \end{pmatrix} W_1^T \quad \text{and} \quad B_2 = Q_2 D_2 W_2^T,$$

where Q_i and W_i are orthogonal matrices of appropriate dimensions, and the D_i are non-negative diagonal matrices. Let $\begin{pmatrix} l_1^T & \lambda_1 \end{pmatrix}$ be the last row of W_1, and let f_2^T be the first row of W_2. Plugging these into (2.5), we get

$$B = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} \begin{pmatrix} D_1 & 0 & 0 \\ \alpha_k l_1^T & \alpha_k \lambda_1 & \beta_k f_2^T \\ 0 & 0 & D_2 \end{pmatrix} \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix}^T \tag{2.6}$$

The middle matrix is quite simple in that its entries can be non-zero only on the diagonal and in the k-th row. Let $S\Sigma G^T$ be the SVD of the middle matrix. Inserting it into (2.6), we get the SVD of B as:

$$B = Q\Sigma W^T \tag{2.7}$$

with

$$Q = \begin{pmatrix} Q_1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & Q_2 \end{pmatrix} S \quad \text{and} \quad W = \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix} G$$

To compute the SVDs of B_1 and B_2, this process can be recursively applied until the sizes of the subproblems are sufficiently small. These small subproblems are then solved using a QR type algorithm (`xBDSQR` in LAPACK). There can be at most $O(\log_2 n)$ levels of recursion.

Gu and Eisenstat [6] found a numerically stable algorithm, which SVD-factorizes the middle matrix from (2.6).

3. Apply back all the Householder transformations to solve the original least squares problem.

   Getting (2.4) and (2.7) together, the solution to the LLS problem can be computed as

   $$\mathbf{x} = VW\Sigma^{-1}Q^T U_1^T \mathbf{b},$$

   where $U_1^T\mathbf{b}$ is computed by applying the Householder transformations directly to $\mathbf{b}$, $Q^T(U_1^T\mathbf{b})$ is computed by applying the Givens rotations to $U_1^T\mathbf{b}$ [6].

3 Algorithms comparison

3.1 Application domain

As follows from the descriptions of the methods, they can be used in different situations. Given the problem in a form of $A\mathbf{x} = \mathbf{b}$:

1. **xGELS** can be used when we are sure the matrix A has a full rank;
2. **xGELSX**, **xGELSS** and **xGELSD** work with any kind of matrix.

3.2 Asymptotic complexity and memory throughput

For a matrix A with m rows and n columns the LSS solution using **xGELS** has a complexity of $2mn^2 - (2/3)n^3$ floating point operations (flops) [2].

For **xGELSY**, **xGELSS** and **xGELSD** the asymptotic complexity is $O(mn^2)$ [3]. It is important to understand that all these algorithms require substantially more operations than a regular QR decomposition, hence have larger coefficients. Comparing these coefficients is considered a part of the numerical experiments (section 4).

3.3 Work memory

`LAPACK` is nice in a way that it specifies the work memory, needed for each algorithm, additionally to the input parameters memory. It is commonly reffered as `work` (memory of the same type as an input data of the length `lwork`) and `iwork` (integer memory of the length liwork). This memory must be allocated by the caller and passed as a separate parameter (or several parameters). This implies that an algorithm itself never allocates memory on heap, which allows us to predict the overall algorithm caching behavior.

There is a minimum memory requirement for each function. For some functions, there is an optimal memory recommendation, the smallest amount of memory for which we get the best performance.

For the list below, we assume that $m \geq n$ and we have a single task (**x** and **b** are vectors). `T` is the input matrices data type. Then the additional work memory needed for each algorithm is:

xGELS	$\texttt{lwork} = 2n \times \textbf{sizeof}(\texttt{T})$
	$\texttt{lwork}_{\texttt{opt}} = (\texttt{nb} + 1)n \times \textbf{sizeof}(\texttt{T})$
xGELSY	$\texttt{lwork} = (4n + 1) \times \textbf{sizeof}(\texttt{T})$
	$\texttt{lwork}_{\texttt{opt}} = (3n + \texttt{nb} * (n + 1)) \times \textbf{sizeof}(\texttt{T})$
xGELSS	$\texttt{lwork} = (3n + \texttt{max}(2n, m)) \times \textbf{sizeof}(\texttt{T})$
xGELSD	$\texttt{lwork} = (13n + 2n * \texttt{smlsiz} + 8n * \texttt{nlvl} + (\texttt{smlsize} + 1)^2) \times \textbf{sizeof}(\texttt{T})$
	$\texttt{liwork} = (3n * \texttt{nlvl} + 11 * n) \times \textbf{sizeof}(\texttt{int})$

where `nb` is the optimum block size, `smlsize` is returned by `ILAENV` and is equal to the maximum size of the subproblems at the bottom of the computation tree (usually about 25), and $\texttt{nlvl} = \lfloor \texttt{log}_2(n/(\texttt{smlsiz}+1)) + 1 \rfloor$. For good performance, `lwork` should generally be larger.

Additionally, different algorithms compute additional data, such as a permutation matrix in `xGELSY` ($n \times$ **sizeof**(`int`)) or singular values in `xGELSS` and `xGELSD` ($n \times$ **sizeof**(`float`) for `SGELSD` and `CGELSD`, and $n \times$ **sizeof**(`double`) for `DGELSD` and `ZGELSD`).

Formulas for generic case matrices can be found in the `LAPACK` documentation [8].

3.4 Performance

Intuitively, the execution times of the algorithms should be compared as follows:

$$time(\textbf{xGELS}) < time(\textbf{xGELSY}) < time(\textbf{xGELSD}) < time(\textbf{xGELSS})$$

Indeed, in the `LAPACK` user guide (section "LAPACK Benchmark") they provide results of the numerical experiments (Figure 1).

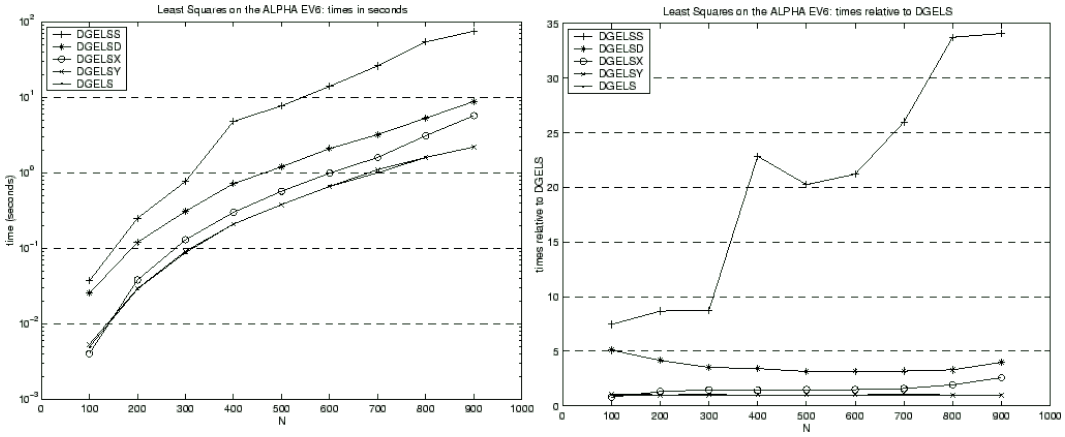

Figure 1: Timings of driver routines for the least squares problem (source: [8]). The left graph shows times in seconds on a Compaq AlphaServer DS-20. The right graph shows times relative to the fastest routine DGELS, which appears as a horizontal line at 1.

3.5 Numerical accuracy

The numerical experiments were performed in [4]. Some of the insights when A is a random matrix:

1. If there is a large gap in the singular values and if p the decay rate of the Picard coefficients is not large then using the QR decomposition with pivoting (`xGELSY`) is, on average, nearly as accurate as the SVD (`xGELSS`).

2. For matrices with small gaps in the singular values the SVD solutions may be, on average, better than QR decomposition with pivoting solutions, but the difference may be modest.

4 Numerical experiments

We ran the following experiments on an Intel® Core™ i9-9880H CPU with 2.3GHz clock, 512KB first level cache, 2MB second level cache and 16MB third level cache. All experiments were run in double precision, i.e. 64-bit, IEEE floating point arithmetic. We let $\epsilon = 2^{-52}$ denote the machine precision.

`LAPACK` and `BLAS` are used from the Intel OneAPI MKL distribution [1]:

```
>> import numpy as np
>> np.show_config()
blas_mkl_info:
    libraries = ['mkl_rt']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
lapack_mkl_info:
    libraries = ['mkl_rt']
    define_macros = [('SCIPY_MKL_H', None), ('HAVE_CBLAS', None)]
...
```

	L1 cache fit	L2 cache fit	L3 cache fit
n = 100	Yes		
n = 200	Yes		
n = 300	No	Yes	
n = 400	No	Yes	
n = 500	No	No	Yes
n = 600	No	No	Yes

Table 1: Shows if the $n \times n$ task fully fits the specified cache.

The high performance code is written in CPython. It accesses the native `LAPACK` functions (Fortran-style pointer arguments) from `scipy.linalg.cython_lapack`. The benchmark data is collected with Python.

4.1 Types of test matrices

We consider solving n-by-n least squares problems with single right hand sides. We use the following test matrices (similar approach to)[7]:

Type 1. These matrices were randomly generated with singular values distributed arithmetically from ϵ to 1: `np.linspace(eps, 1, n)`;

Type 2. These matrices were randomly generated with singular values distributed geometrically from ϵ to 1: `np.logspace(eps, 1, n, base=2)-1`;

Type 3. These matrices have 1 singular value at 1 and the other $n-1$ clustered at ϵ.

We present square problems only, since m-by-n problems with $m \gg n$ are generally reduced to an n-by-n problem by an initial QR decomposition, and this dominates all later computations.

The matrix sizes were chosen so that the tasks (input matrices and optimal work memory together) fall in different caches (Table 1).

For each test, we select 10 random matrices and run the algorithm 100 times for each matrix, and compute the average time of 1000 runs.

4.2 Performance of algorithms with limited memory

The first experiment includes running all functions with the minumum memory allocated, as described in the Section 3.3. The results can be observed in the Table 2.

Table 2: MKL LAPACK, limited memory. Singular values distributed arithmetically from ϵ up to 1 (Type 1). Average time in `ms` of each function w.r.t. the matrix size ($n \times n$):

Function\n	100	200	300	400	500	600
DGELS	0.141	0.39	0.865	1.712	2.955	4.658
DGELSY	0.245	2.279	7.45	16.804	29.322	47.059
DGELSS	2.502	10.858	24.826	50.779	87.234	140.415
DGELSD	1.248	4.484	10.116	17.139	26.886	39.602

Right away we can see that the results of the `LAPACK` benchmark, presented in the Figure 1, appear to perfectly align with the observed results.

We can also observe the nonlinear caching nature of algorithms:

- `DGELS`: 200x200 matrix is 2.7 times slower than 100x100 (x4 matrix entries), 300x300 matrix is 2.2 times slower than 200x200 (x2.25 matrix entries), 600x600 matrix is 1.57 times slower than 500x500 (x1.44 matrix entries). So it's more like $O(n^2)$;
- `DGELSY`: things are completely different: 200x200 matrix is 9.3 times slower than 100x100, and we can clearly see the $O(n^3)$ asymptotic.
- `DGELSS` is the slowest of all algorithms, and asymptotically it's a disaster.

4.3 Performance of algorithms with optimal memory

Let's run the same Type 1 test but allocate more `work` memory for each algorithm. Each algorithm was asked for the memory they want. At average, `DGELS` requested x20 more memory comparing to minimal, `DGELSY` requested x8 more memory, `DGELSS` requested x30 more memory, `DGELSD` was fine with the initial estimate, but this initial estimate is almost the same as the optimal memory amount for `DGELSS`.

The next three tables present benchmark results for all three test matrix types:

Table 3: MKL LAPACK, optimal memory. Singular values distributed arithmetically from ϵ up to 1 (Type 1). Average time in `ms` of each function w.r.t. the matrix size ($n \times n$):

Function\n	100	200	300	400	500	600
DGELS	0.148	0.389	0.895	1.763	2.955	4.747
DGELSY	0.265	1.162	2.587	4.701	7.797	12.215
DGELSS	2.538	8.587	16.065	28.485	46.015	72.369
DGELSD	1.313	4.55	10.217	17.109	26.612	40.172

From the Type 1 benchmark we can notice a significant performance gain for `DGELSY` and `DGELSS`, comparing to the minimum memory benchmark. `DGELS` gain is statistically insignificant.

Table 4: MKL LAPACK, optimal memory. Singular values distributed geometrically from ϵ up to 1 (Type 2). Average time in `ms` of each function w.r.t. the matrix size ($n \times n$):

Function\n	100	200	300	400	500	600
DGELS	0.137	0.374	0.885	1.714	2.981	4.706
DGELSY	0.251	1.133	2.621	4.664	7.748	11.936
DGELSS	2.289	8.613	15.641	27.742	44.161	70.233
DGELSD	1.208	4.534	9.787	17.009	25.96	38.063

When singular values are distributed geometrically (Type 2 test), we don't observe significant changes, comparing to Type 1.

The next type (Type 3) is special in a way, that we deal with a matrix of extremely low rank. That should trigger SVD algorithms to stop early.

Also, we should notice that `DGELS` is skipped for this test.

Table 5: MKL LAPACK, optimal memory. 1 singular value at 1 and the other clustered at ϵ (Type 3). Average time in `ms` of each function w.r.t. the matrix size ($n \times n$):

Function\n	100	200	300	400	500	600
DGELS	-	-	-	-	-	-
DGELSY	0.335	1.506	3.537	6.303	10.661	16.191
DGELSS	2.949	10.837	19.626	35.594	58.918	97.404
DGELSD	0.479	2.165	5.323	9.201	14.972	22.356

Indeed, `DGELSD` was more than 2 times faster, comparing to the full-rank tests. Interestingly, `DGELSS` showed worse results.

4.4 Fortran LAPACK

We can also use a regular `Fortran LAPACK` implementation and measure the functions time for the Type 1 test:

Table 6: Fortran LAPACK, optimal memory. Singular values distributed arithmetically from ϵ up to 1 (Type 1). Average time in `ms` of each function w.r.t. the matrix size ($n \times n$):

Function\n	100	200	300	400	500	600
DGELS	1.362	5.309	9.526	16.776	27.182	39.609
DGELSY	1.388	14.734	27.16	44.437	69.252	99.98
DGELSS	11.407	52.523	110.305	222.904	379.462	608.879
DGELSD	5.061	34.559	64.708	108.004	175.674	257.156

The timings are 7-8 slower, as expected, because MKL parallelizes the algorithms.

5 Conclusions

We have presented 4 possible options to solve linear least square problems using `LAPACK`. The algorithms are described in terms of the implementation, as well as the memory requirements and throughput. The algorithms comparison, pros and cons are discussed, too.

Numerical experiments to measure performance of the algorithms are performed. **xGELS** should be preferred for the full-rank tasks, it is 2x faster than `xGELSY` and 5x faster than `xGELSD`, and needs the smallest amount of memory. For rank-deficient matrices, **xGELSY** should be preffered. In case of ill-formed or low-rank matrices **xGELSD** might give more accurate results, it can also be considered as a generic approach, but needs more work memory than the previously mentioned functions. **xGELSS** should not be used.

References

[1] Developer Reference for Intel® oneAPI Math Kernel Library - C. `https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html`, 2020. [Online; accessed 7-February-2021].

[2] Stephen Boyd and Lieven Vandenberghe. *Introduction to applied linear algebra: vectors, matrices, and least squares*. Cambridge university press, 2018.

[3] Leslie Foster and Rajesh Kommu. Algorithm 853: An efficient algorithm for solving rank-deficient least squares problems. *ACM Transactions on Mathematical Software (TOMS)*, 32(1):157–165, 2006.

[4] Leslie V Foster. Why the qr factorization can be more accurate than the svd, May 2004.

[5] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[6] Ming Gu and Stanley C Eisenstat. A divide-and-conquer algorithm for the bidiagonal svd. *SIAM Journal on Matrix Analysis and Applications*, 16(1):79–92, 1995.

[7] Ming Gu, James Demmely, and Inderjit Dhillonz. Efficient computation of the singular value decomposition with applications to least squares problems. Technical report, Citeseer, 1994.

[8] Univ. of Tennessee; Univ. of California Berkeley; Univ. of Colorado Denver; and NAG Ltd. LAPACK — Linear Algebra PACKage. `http://performance.netlib.org/lapack/`, 2019. [Online; accessed 7-February-2021].