Computing Large Scale Linear Least Squares with SKiLLS

Zhen Shao

December 16, 2020

1 Overview

SKiLLS (SKetchIng-Linear-Least-Sqaures) is a C++ package for finding solutions to over-determined linear least square problems. SKiLLS uses a modern dimensionality reduction technique called sketching, and is particularly suited for large scale linear least squares where the number of measurements/observations is far greater than the number of variables.

Mathematically, SkiLLS solves

$$\min_{x \in \mathbb{R}^d} f(x) := \|Ax - b\|_2^2, \tag{1}$$

where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$ given. The matrix A is allowed to be rank-deficient or nearly rank-deficient.

1.1 When to use SKiLLS

If A in (1) is dense, the state-of-the-art sketching solver is Blendenpik [2]. Comparing to solvers in the classical state-of-the-art numerical package LAPACK [1], Blendenpik is two times faster on matrices of size $20,000 \times 500$ and four times faster on matrices of size $100,000 \times 2500$. Comparing to classical iterative solver LSQR [7], Blendenpik is 80 times faster on matrices of size $20,000 \times 1,000$ with condition number 100.

Blendenpik only solves (1) when the matrix A has full numerical rank, as illustrated in Table 1. Two solvers are included in the library SKiLLS for dense A. The robust version, *ls_dense_hashing_blendenpik* solves problems 1 when A is numecially rank-deficient but is as fast as Blendenpik when A has full rank (takes 70% to 130% time on matrices of different sizes). The fast version, *ls_dense_hashing_blendenpik_noCPQR* is 1.6 times faster than Blendenpik on coherent matrices[1] of size $40,000 \times 4,000$ and $90,000 \times 2250$, and slightly faster than Blendenpik on other types of input (about 1.1 times faster).

If A in (1) is sparse, the state-of-the-art solvers are SPQR [3] and HSL [8], which uses sparse QR factorization of A and LSQR with an incomplete Cholesky preconditioner, respectively. Our library has a single sparse solver, *ls_sparse_spqr*. It significantly outperforms both state-of-the-art sparse solvers on large random sparse ill-conditioned matrices. It is 10 times faster than HSL and 7 times faster than SPQR on $120,000 \times 5,000$ random sparse matrices with 1% non-zero entries and condition number equals to 10^6. And it is 3 times faster than both HSL and SPQR on $40,000 \times 2,000$ matrices with 1% non-zero entries and condition number equals to 10^6.

[1] Matrices of the form $A \in \mathbb{R}^{n \times d} = \binom{I_{d \times d}}{0} + 10^{-8} J_{n,d}$ where $J_{n,d} \in \mathbb{R}^{n \times d}$ is a matrix of all ones.

Our sparse solver also performs extremely well on highly over-determined sparse inputs from real applications. It is the fastest solver on more than 75% of problems in the Florida Matrix Collection [5] with the matrix A defined in (1) having $n \geq 30d$ and $n \geq 20,000$. It is also the fastest solver on more than 90% of problems with the matrix A having $n \geq 10d$, $n \geq 20,000$ but additionally having more than 1% of non-zero entries.

Because sketching exploits redundancy of rows, the performance gain compared to non-sketching solvers[2] will be larger on A with larger n/d radio.

For further details of computational advantages, see our paper. *Computing Large Scale Linear Least Squares with SKiLLS, C. Cartis and Z. Shao.*

2 Installing SKiLLS

2.1 Dependency

2.2 Installation instruction

2.3 Test the installation

3 Using SKiLLS

This is a library of three routines for solving problem (1).

- If the matrix A is dense, *ls_ dense_ hashing_ blendenpik* uses LSQR with a preconditioner built from a sketch of the matrix A using Hashed-Randomised-Hadamard-Transform (HRHT) and Randomised-Column-Pivoted-QR (RCPQR) to solve (1).

- If A is dense and has full numerical rank, *ls_ dense_ hashing_ blendenpik_ noCPQR* uses LSQR with a preconditioner built from a sketch of the matrix A using HRHT and Column-Pivoted-QR (CPQR) to solve (1).

- If A is sparse, *ls_ sparse_ spqr* uses LSQR with a preconditioner built from a sketch of the matrix A using $s-$hashing and sparse QR (SPQR) to solve (1).

3.1 Data structure

Dense Vector and Matrix SKiLLS uses C++ class Vec for dense vectors and C++ class Mat for dense matrices. To create a $n \times 1$ vector from already existed data, take a pointer to the data and call Vec(n, *data). To create a $n \times d$ matrix from already existed data, take a pointer to the data and call Mat(n, d, *data). Basic matrix operations such as return a specific row, column, or matrix-matrix and matrix vector multiplications are implemented. These classes are defined in include/Vec.hpp and include/impl/Mat.hpp respectively.

Sparse Matrix SKiLLS uses Compressed Column Data C structure for sparse matrices and vectors. The sparse data structure is from SuiteSparse [4]. The cs_dl and cholmod_sparse structures are defined in cs.h and cholmod_core.h respectively the in SuiteSparse/include folder.

[2] LAPACK, HSL, SPQR

3.2 Dense problems

To compute a solution of a linear least square problem where the matrix A is dense, a call of the following form should be made.

ls_dense_hashing_blendenpik(A, b, x, rank, flag, it, gamma, k, abs_tol, rcond, it_tol, max_it, debug, wisdom):

- Inputs
  1. **A** is type Mat_d containing a $\mathbb{R}^{m\times n}$ matrix defined in (1).
  2. **b** is type Vec_d containing a $\mathbb{R}^{m}$ vector defined in (1).
- Main Outputs
  1. **x** is type Vec_d containing a $\mathbb{R}^{n}$ vector which is a solution of (1).
- Auxillary Outputs
  1. **rank** is a scalar containing the detected numerical rank of the matrix A.
  2. **flag** is a scaler. flag=0 indicates LSQR has converged. flag=1 indicates LSQR has not converged.
  3. **it** is a scaler indicating the number of LSQR iterations taken.
- Parameters
  1. **gamma** is the over-sampling ratio used in sketching. Bigger gamma typically gives higher quality preconditioner but slower running time. The default value of gamma is 1.7.
  2. **k** is number of non-zeros per column in the hashing matrix as part of sketching. Bigger k typically gives higher quality preconditioner but slower running time. The default value of k is 1.
  3. **abs_tol** specifies an absolute residual tolerance for the solution x of 1. If the algorithm finds an x satisfies $\|Ax-b\|_2 \leq abs_tol$, it terminates and returns x. The default value is 10^{-8}
  4. **it_tol** specifies the relative residual tolerance for LSQR convergence, see [7]. The default value is 10^{-6}.
  5. **max_it** specifies the maximum iteration of LSQR, the default value is $10,000$.
  6. **debug** is a flag. If debug=1, the solver prints additional outputs for diagosis.
  7. **wisdom** is a flag. If wisdom=1, the macro variable FFTW_WISDOM_FILE in inclde/-Config.hpp must be defined as the path to a FFTW wisdom file, see fftw documentation at fftw.org. If wisdom=0, the solver does not use FFTW wisdom and may run slower.

If the matrix A is known to be of full numerical rank, then a different routine can be used which is slightly faster:

ls_dense_hashing_blendenpik_noCPQR(A, b, x, rank, flag, it, gamma, k, it_tol, max_it, debug, wisdom). The arguments usage is the same as above, but we don't need the rank-detection parameter rcond, and the shortcut related to abs_tol is not implemented.

Example The following program solves an example of problem (1) with given A and b.

```
#include <iostream>
#include "Config.hpp"
#include "SpMat.hpp"
#include "Random.hpp"
#include <math.h>
#include "RandSolver.hpp"
#include "cs.h"
#include <stdio.h>
#include "SuiteSparseQR.hpp"
#include "cholmod.h"
#include "lsex_all_in_c.h"
#include "IterSolver.hpp"
#include <sys/timeb.h>
#include "bench_config.hpp"
#include "blendenpik.hpp"

int main(int argc, char **argv)
{
        // create data
        long m = 4;
        long n = 2;
        double data_A[8] = {0.306051, 1.53078,1.64493, 1.61322,
         0.2829,0.474476, 0.586278, 0.610202};

        double data_b[4] = {0.0649338,0.845946, 0.0164085,0.247119};

        Mat_d A(m,n,data_A);
        Vec_d b(m,data_b);

        // parameters
        long k = NNZ_PER_COLUMN;
        double gamma = OVER_SAMPLING_RATIO;
        long max_it = MAX_IT;
        double it_tol = IT_TOL;
        double rcond = 1e 10;
        int wisdom = 0;
        double abs_tol = ABS_TOL;

        // storage

        long it;
        int flag;
        long rank;
        double residual;
        Vec_d x(A.n());

        // solve
        ls_dense_hashing_blendenpik(
        A, b, x, rank, flag, it, gamma, k, abs_tol,
        rcond, it_tol, max_it, debug, wisdom);

        std::cout << "A is: "<< A << std::endl;
        std::cout << "b is: "<< b << std::endl;
        std::cout << "solution is: "<< x << std::endl;

        // compute residual
        A.mv('n', 1, x,  1, b);
```

```
        std::cout << "Residual of ls_blendenpik_hashing is: "<<  nrm2(b) << std::endl;
        std::cout << "Iteration of ls_blendenpik_hashing is: " << it << std::endl;

}
```

3.3 Sparse problems

To compute a solution of a linear least square problem where the matrix A is sparse, a call of the following form should be made.

ls_sparse_spqr(*A, b, x, rank, flag, it, gamma, k, abs_tol, ordering, it_tol, max_it, rcond, peturb, debug):

- Inputs
  1. **A** is pointer to type cs_dl containing a $\mathbb{R}^{m\times n}$ sparse matrix defined in (1).
  2. **b** is type Vec_d containing a $\mathbb{R}^{m}$ vector defined in (1).
- Main Outputs
  1. **x** is type Vec_d containing a $\mathbb{R}^{n}$ vector which is a solution of (1).
- Auxillary Outputs
  1. **rank** is a scalar containing the detected numerical rank of the matrix A.
  2. **flag** is a scaler. flag=0 indicates LSQR has converged. flag=1 indicates LSQR has not converged.
  3. **it** is a scaler indicating the number of LSQR iterations taken.
- Parameters
  1. **gamma** is the over-sampling ratio used in sketching. Bigger gamma typically gives higher quality preconditioner but slower running time. The default value of gamma is 1.7.
  2. **k** is number of non-zeros per column in the hashing matrix as part of sketching. Bigger k typically gives higher quality preconditioner but slower running time. The default value of k is 1.
  3. **abs_tol** specifies an absolute residual tolerance for the solution x of 1. If the algorithm finds an x satisfies $\|Ax-b\|_2 \leq abs_tol$, it terminates and returns x. The default value is 10^{-8}
  4. **ordering** is an integer specifying a fill-reduced ordering for sparse QR factorization. The default value is SPQR_ORDERING_DEFAULT. See SuiteSparseQR documentation for more information.
  5. **it_tol** specifies the relative residual tolerance for LSQR convergence, see [7]. The default value is 10^{-6}.
  6. **max_it** specifies the maximum iteration of LSQR, the default value is $10,000$.
  7. **rcond** is numerical-ill-conditioning tolerance for the preconditioner returned by sparse QR. If the condition number of the preconditioner is larger than $1/rcond$, a warning will be printed. The default value of rcond is 10^{-12}
  8. **perturb** is a real number. It has no-effect on the algorithm and is part of an ongoing development.
  9. **debug** is a flag. If debug=1, the solver prints additional outputs for diagosis.

Example The following program solves an example of problem (1) with given sparse A and dense b.

```
#include <iostream>
#include "Config.hpp"
#include "SpMat.hpp"
#include "Random.hpp"
#include <math.h>
#include "RandSolver.hpp"
#include <time.h>
#include "cs.h"
#include <stdio.h>
#include "SuiteSparseQR.hpp"
#include "cholmod.h"
#include "lsex_all_in_c.h"
#include "IterSolver.hpp"
#include <sys/timeb.h>
#include "bench_config.hpp"

int main(int argc, char const *argv[])
{
        // Create a sparse matrix in compressed column format
    long m = 4;
    long n = 3;
    long nnz = 5;

    long col[4] = {0,  1,  3,  5};
    long row[5] = {0,  0,  2,  1,  3};
    double val[5] = {2.0,  3.0,  1.0,  4.0,  5.0};

    cholmod_common Common, *cc;
    cc = &Common;
    cholmod_l_start(cc);
    cc >print = 4;

    cholmod_sparse* A;
    A = cholmod_l_allocate_sparse(
    m, n, nnz, true, true, 0, CHOLMOD_REAL, cc);

    A >p = col;
    A >i = row;
    A >x = val;
    cholmod_l_print_sparse( A, "A", cc);
    cholmod_l_finish(cc);


        // input
        cs_dl *A;
    A = to_cs_dl(Acholmod);

    double *b_data = (double*)malloc((A >m)*sizeof(double));
    for (long i=0; i<A >m; i++){
        b_data[i] = 1;
    }
    Vec_d b(A >m, b_data);

    // parameters
    long k = NNZ_PER_COLUMN;
    double gamma = OVER_SAMPLING_RATIO;
```

```
    long max_it = MAX_IT;
    double it_tol = IT_TOL;
    double abs_tol = ABS_TOL;
    double rcond = 1e 12;
    int ordering = SPQR_ORDERING;

    // storage

    long it;
    int flag;
    long rank;
    double t_start;
    double t_finish;
    double residual;
    Vec_d x_ls_qr(A >n);

    // solve
    ls_sparse_spqr(*A, b, x_ls_qr,
    rank, flag, it, gamma, k, abs_tol, ordering, it_tol, max_it, RCOND_THRESHOLD, PERTURB, debug);

    CSC_Mat_d A_CSC(A >m, A >n, A >nzmax,
        (long*)A >i, (long*)A >p, (double*)A >x);
    A_CSC.mv('n', 1, x_ls_qr, 1, b);

    std::cout << "Residual of ls_sparse_spqr is: "<< nrm2(b) << std::endl;
        return 0;
}
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[2] H. Avron, P. Maymounkov, and S. Toledo. Blendenpik: Supercharging LAPACK's Least-Squares Solver. *SIAM Journal on Scientific Computing*, 32(3):1217–1236, Jan. 2010.

[3] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):8, Nov. 2011.

[4] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal Multithreaded Rank-Revealing Sparse QR Factorization. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), Dec. 2011.

[5] T. A. Davis and Y. Hu. The university of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1–25, Nov. 2011.

[6] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), Dec. 2011.

[7] C. C. Paige and M. A. Saunders. LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares. *ACM Transactions on Mathematical Software*, 8(1):43–71, Mar. 1982.

[8] J. Scott and M. Tůma. HSL_MI28: An Efficient and Robust Limited-Memory Incomplete Cholesky Factorization Code. *ACM Transactions on Mathematical Software*, 40(4):1–19, June 2014.

A Numerical Results

	lp_ship12l	Franz1	GL7d26	cis-n4c6-b2	lp_modszk1	rel5	ch5-5-b1
SVD	18.336	26.503	50.875	6.1E-14	33.236	14.020	7.3194
Blendenpk	NaN	9730.700	NaN	3.0E+02	NaN	NaN	340.9200
Ski-LLS-dense	18.336	26.503	50.875	5.3E-14	33.236	14.020	7.3194
	n3c5-b2	ch4-4-b1	n3c5-b1	n3c4-b1	connectus	landmark	cis-n4c6-b3
SVD	9.0E-15	4.2328	3.4641	1.8257	282.67	1.1E-05	30.996
Blendenpk	1.3E+02	66.9330	409.8000	8.9443	NaN	NaN	3756.200
Ski-LLS-dense	5.2E-15	4.2328	3.4641	1.8257	282.67	1.1E-05	30.996

Table 1: Residual of solvers for a range of approximate rank-deficient problems taken from the Florida matrix collection [6]. We see while Blendenpik struggles, SKiLLS achives the same residual accuracy as SVD method.