Solving Large Scale Linear Least Squares with Ski-LLS

Zhen Shao

May 27, 2021

1 Overview

Ski-LLS (SKetchIng-Linear-Least-Squares) is a C++ package for finding solutions to over-determined linear least square problems. Ski-LLS 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, Ski-LLS 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 Ski-LLS

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 [8], Blendenpik is 80 times faster on matrices of size $20,000 \times 1,000$ with condition number 100. However, Blendenpik only solves (1) when the matrix A is dense, and has full numerical rank. Ski-LLS improves upon Blendenpik on both robustness and speed. Two solvers are included in the library Ski-LLS for dense A. The robust version, `ls_dense_hashing_blendenpik` solves problems (1) when A is numerically 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 [5] and HSL [9], 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 [6] 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 [4]. Also see our conference paper [3].

2 Installing Ski-LLS

Ski-LLS can be downloaded from `https://github.com/numericalalgorithmsgroup/Ski-LLS` and it is distributed under the BSD license. Please follow the following installation instruction after downloading the package.

2.1 Installing dependencies and configuration

Ski-LLS relies on the following external packages: LAPACK & BLAS, SuiteSparse, FFTW, and BOOST C++ library. It is necessary to point `Makefile` to the correct location of the compiled libraries and include directories with the appropriate header files via make variables `LIBS_LAPACK, LIBS_SPARSE, SPARSE_INCLUDE, LIBS_FFTW, FFTW_INCLUDE` and `BOOSTROOT` respectively. For your convenience, all these make variables are located in a configuration makefile and two sample configurations for Linux/Mac are provided in `./config` directory. The first one (`./config/make_gcc.inc`) assumes that there are no pre-installed libraries available and all of them are compiled locally with the GNU Compiler Collection. The second one (`./config/make_intel.inc`) shows how to benefit from Intel compiler and their optimized Math Kernel Library (MKL). Note that the current version of Ski-LLS is using 64-bit integer size (ILP64) so LAPACK, BLAS and SuiteSparse needs to be compatible. The following paragraphs provides a step-by-step guide on how to compile them and set everything up. It should also help as a reference should you wish to use a different setting or system.

Using the reference version of LAPACK and GNU Compilers The following steps show how to compile all libraries from scratch. The reference version of LAPACK and BLAS is used. This works great as a reference point, however, **it is highly recommended to use tuned version of BLAS** (such as GotoBLAS or MKL) to achieve a good performance suitable for production or benchmarking.

1. Reference (vanilla) LAPACK installation.
   First download and unpack the LAPACK.

   ```
   wget -nd https://github.com/Reference-LAPACK/lapack/archive/v3.9.0.tar.gz
   tar -xzf v3.9.0.tar.gz
   cd lapack-3.9.0
   ```

   Then, modify LAPACK's compilation flags to use 64 bits integers. First make a copy of the default configuration

   ```
   cp make.inc.example make.inc
   ```

[2] LAPACK, HSL, SPQR

and change the values of the following three variables in `make.inc`

```
CFLAGS = -O3 -fPIC
FFLAGS = -O2 -frecursive -fPIC -fdefault-integer-8
FFLAGS_NOOPT = -O0 -frecursive -fPIC -fdefault-integer-8
```

Lastly, compile LAPACK

```
make -j 20 blaslib lapacklib
```

This should generate two libraries: `lapack-3.9.0/liblapack.a, librefblas.a`. Please point `LIBS_LAPACK` in `config/make_gcc.inc` to them in such a way that they can be linked to C/C++ code, in this case Fortran runtime library (`-lgfortran`) needs to be added, e.g.

```
LAPACKROOT = /fserver/zhens/testInstall/Dependencies/lapack-3.9.0
LIBS_LAPACK = ${LAPACKROOT}/liblapack.a ${LAPACKROOT}/librefblas.a -
    lgfortran
```

2. SuiteSparse installation.
   First download and unpack SuiteSparse

```
wget -nd https://github.com/DrTimothyAldenDavis/SuiteSparse/archive/v5.6.0.
    tar.gz
tar -xzf v5.6.0.tar.gz
cd SuiteSparse-5.6.0
```

Then compile SuiteSparse by calling the following command; note that we need to point to our newly compiled LAPACK and BLAS libraries (adjust the path accordingly) and specify that 64-bit integers are used:

```
make CC=gcc CXX=g++ BLAS="/fserver/zhens/testInstall/Dependencies/lapack
    -3.9.0/librefblas.a -lgfortran" LAPACK=/fserver/zhens/testInstall/
    Dependencies/lapack-3.9.0/liblapack.a CHOLMOD_CONFIG=-DLONGBLAS=long
    UMFPACK_CONFIG=-DLONGBLAS=long
```

After successfully installing SuiteSparse, point the variables `LIBS_SPARSE` and `SPARSE_INCLUDE` in `config/make_gcc.inc` to the correct location, e.g.

```
SUITESPARSEROOT = /fserver/zhens/testInstall/Dependencies/SuiteSparse
    -5.6.0/
SPARSE_INCLUDE = ${SUITESPARSEROOT}/include
LIBS_SPARSE = -L${SUITESPARSEROOT}/lib -lcxsparse -lcholmod -lspqr
```

SuiteSparse by default produces shared libraries so system variable `LD_LIBRARY_PATH` needs to be created or extended for the executables to find SuiteSparse libraries. For example, if you are using bash shell call

```
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:$PWD/lib"
```

and similarly if you are using csh shell

```
setenv LD_LIBRARY_PATH "${LD_LIBRARY_PATH}:$PWD/lib"
```

3. FFTW installation.
   Download, unpack and compile FFTW:

```
wget -nd http://www.fftw.org/fftw-3.3.8.tar.gz
tar -xzf fftw-3.3.8.tar.gz
cd fftw-3.3.8
configure
make
```

Then set LIBS_FFTW, FFTW_INCLUDE in config/make_gcc.inc as appropriate, e.g.

```
FFTWROOT = /fserver/zhens/testInstall/Dependencies/fftw-3.3.8/
FFTW_INCLUDE = ${FFTWROOT}/api
LIBS_FFTW = ${FFTWROOT}/.libs/libfftw3.a
```

4. BOOST C++ library installation.
   BOOST C++ library is a header only library, therefore does not require compilation. Download and unpack BOOST C++ library:

```
wget -nd https://boostorg.jfrog.io/artifactory/main/release/1.72.0/source/
    boost_1_72_0.tar.gz
tar -xzf boost_1_72_0.tar.gz
```

   Then set BOOSTROOT in config/make_gcc.inc to the correct location, e.g.

```
BOOSTROOT ?= /fserver/zhens/testInstall/Dependencies/boost_1_72_0/
```

5. (optional) Adjust compilers and compiler flags (such as optimization level) via standard make variables CC, CFLAGS, CXX and CXXFLAGS in the configuration makefile, config/make_gcc.inc uses the following:

```
CXX = g++
CC = gcc
CXXFLAGS = -Wall -O3 -march=native -fPIC -std=c++11
CFLAGS = -Wall -O3
```

Using Intel's tuned version of LAPACK (MKL) and Intel compilers The following steps demonstrate how to set up Ski-LLS using Intel MKL, this configuration has been used for benchmarking reported in the paper [4].

1. Configuration for MKL LAPACK & BLAS
   Check that the environmental variable $MKLROOT is set in your shell. If not, you might need to call a command such as

```
source /fserver/intel/opt/intel/compilers_and_libraries_2019.1.144/linux/
    bin/compilervars.sh intel64
```

   for bash or a similar one for csh

```
source /fserver/intel/opt/intel/compilers_and_libraries_2019.1.144/linux/
    bin/compilervars.csh intel64
```

   Then set LIBS_LAPACK in config/make_intel.inc to the appropriate MKL libraries, for example for dynamic linking:

```
LIBS_LAPACK = -L${MKLROOT}/lib/intel64 -Wl,--no-as-needed -lmkl_intel_ilp64
    -lmkl_sequential -lmkl_core -lpthread -lm -ldl
```

   Note that we need to use 64-bit integers (ILP64) version of LAPACK and BLAS. The easiest is to check the MKL Link Advisor[3] for your version of MKL and operating system.

2. Install SuiteSparse.
   Download and unpack SuiteSparse as before, but point make to the MKL version of LAPACK & BLAS (and the usage of 64-bit integer size as previously):

[3] https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl/link-line-advisor.html

```
make CC=icc CXX=icc BLAS="-L${MKLROOT}/lib/intel64 -Wl,--no-as-needed -
    lmkl_intel_ilp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl" LAPACK=
    "" CHOLMOD_CONFIG=-DLONGBLAS=long UMFPACK_CONFIG=-DLONGBLAS=long
```

Set `LIBS_SPARSE` and `SPARSE_INCLUDE` in `config/make_intel.inc` to the correct location. If you are using shared SuiteSparse libraries, don't forget to adapt system variable `LD_LIBRARY_PATH`, similarly as before.

3. FFTW library installation is identical to the previous case, just modify the location in `FFTWROOT` in `config/make_intel.inc` as appropriate.

4. BOOST C++ library installation is identical to the previous case, just modify the location in `BOOSTROOT` in `config/make_intel.inc` as appropriate.

5. (optional) Compilers and their flags can be modified, `config/make_intel.inc` uses

```
CXX = icc
CC = icc
CXXFLAGS =  -DMKL_ILP64  -m64  -I"${MKLROOT}/include" -Wall -O3 -march=
    native -fPIC -std=c++11
CFLAGS = -O3
```

2.2 Compilation and testing of Ski-LLS

If the configuration makefile is correctly adapted (as described in the previous section), it is sufficient to point `make` to it by setting `CONFIG=config/my_config_file` on the command line and invoke the desired target. If no `CONFIG` is given, `config/make_gcc.inc` is assumed. Calling `make` without any target with print help showing all possibilities.

Typically, you will invoke `make` in the top-level directory of the Ski-LLS package with target `lib` to build the Ski-LLS package, for example:

```
make CONFIG=config/make_intel.inc lib
```

To test the installation by running all the solvers (the three solvers in Ski-LLS, along with the others that are compared in our paper) on several small data files included in the distribution, use:

```
make CONFIG=config/make_intel.inc test
```

To compile, link and run the minimal working example of solvers provided in Ski-LLS, type

```
make CONFIG=config/make_intel.inc example
```

To use Ski-LLS package outside of the provided driver files and make, the source will need to include the main header file of the package `#include "ski-lls.h"`, compile the source while pointing to the Ski-LLS include directory, e.g. `-Iski-lls/include` and link against all necessary libraries `ski-lls/LIB/libski-lls.a $(LIBS_SPARSE) $(LIBS_FFTW) $(LIBS_LAPACK) -lpthread -lm -ldl`. Don't forget to adapt `LD_LIBRARY_PATH` to reflect any dynamic linking, such as of SuiteSparse.

2.3 Optional tuning of FFTW

Using FFTW wisdom means pre-tuning FFTW for better performance. Note that this does not affect the sparse solver in Ski-LLS. If one does not wish to use FFTW wisdom:

1. Put the wisdom argument=0 whenever present (see the next section), in the solver routine. This does not affect compilation. By default, the package does not assume FFTW wisdom is built and therefore sets the macro WISDOM=0 in /include/bench_config.hpp.

If one wishes to use FFTW wisdom:

1. Config.hpp has two macros, FFTW_TIMES, FFTW_QUANT, that will be used for testbuild_fftw_wisdom.cpp. The build_fftw_wisdom will try executing fftw with input sizes = linspace(FFTW_QUANT, FFTW_QUANT*FFTW_TIMES, FFTW_QUANT), where linspace(start, finish, step) is a linear space.

2. The user then needs to compile build_fftw_wisdom using the Makefile in test/.

3. After compilation, build_fftw_wisdom.out accepts two arguments: (0 | 1 | 2 | 3 , string), where the first argument is the calibration level with 0 being the lowest and 3 being the highest, the second argument is the directory path to the generated wisdom file.

4. Then, the user needs to go back to Config.hpp, put the path of the generated wisdom file into the macro FFTW_WISDOM_FILE, and define FFTW_WISDOM_FLAG by
   | f==0 = FFTW_ESTIMATE
   | f==1 = FFTW_MEASURE
   | f==2 = FFTW_PATIENT
   | f==3 = FFTW_EXHAUSTIVE
   where f = first argument given to build_fftw_wisdom.out in Step 3.

5. Set the wisdom argument=1 whenever present.

3 Using Ski-LLS

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

- `ls_dense_hashing_blendenpik` is suitable if the matrix A is dense, it uses LSQR [8] with a preconditioner built from a sketch of the matrix A using Hashed-Randomised-Discrete-Hartley-Transform (HR-DHT)[4] and Randomised-Column-Pivoted-QR (R-CPQR) [7] to solve (1).

- `ls_dense_hashing_blendenpik_noCPQR` is a version if A is dense and has full numerical rank, it uses LSQR with a preconditioner built from a sketch of the matrix A using HR-DHT and QR to solve (1).

- `ls_sparse_spqr` is aimed for sparse matrices A and uses LSQR with a preconditioner built from a sketch of the matrix A using $s-$hashing and sparse QR (SPQR) [5] to solve (1).

3.1 Data structure

Dense Vector and Matrix Ski-LLS 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/Mat.hpp` respectively.

[4] A matrix F defined by $F_{ij} = \sqrt{1/n}\,[\cos(2\pi(i-1)(j-1)/n) + \sin(2\pi(i-1)(j-1)/n)]$.

Sparse Matrix Ski-LLS uses the Compressed Column data structure for sparse matrices and vectors. The sparse data structure is from SuiteSparse [5]. The `cs_dl` and `cholmod_sparse` structures are defined in `cs.h` and `cholmod_core.h` respectively in `SuiteSparse/include` folder.

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 $\mathbb{R}^{n \times d}$ matrix defined in (1).
  2. **b** is type Vec_d containing a $\mathbb{R}^{n}$ vector defined in (1).
- Main Outputs
  1. **x** is type Vec_d containing a $\mathbb{R}^{d}$ 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 scalar. flag=0 indicates LSQR has converged. flag=1 indicates LSQR has not converged.
  3. **it** is a scalar 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, chosen by calibration in our paper [4].
  2. **k** is the 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, chosen by calibration [4].
  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. The default value is 10^{-6}.
  5. **rcond** (default value is 10^{-12}), which is a parameter used to determine the numerical rank of A. See [4] for detail.
  6. **max_it** specifies the maximum iteration of LSQR, the default value is $10,000$.
  7. **debug** is a flag. If debug=1, the solver prints additional outputs for diagosis.
  8. **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. Also see the installation section.

If the matrix A is known to be of full numerical rank, then a different routine can be used which is 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 (`test/ski-llsDenseExample.cpp`) solves an example of problem (1) with given A and b, where

$$A = \begin{pmatrix} 0.306051 & -1.53078 \\ 1.64493 & -1.61322 \\ -0.2829 & 0.474476 \\ -0.586278 & -0.610202 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 0.0649338 \\ 0.845946 \\ -0.0164085 \\ 0.247119 \end{pmatrix}. \tag{2}$$

```
#include <iostream>
#include "ski-lls.h"

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;
        int debug = 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;
    // The correct value is x = (-0.2122, 0.720)

}
```

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_thres, peturb, debug)`

- Inputs
  1. **A** is pointer to type cs_dl containing a $\mathbb{R}^{n\times d}$ sparse matrix defined in (1).
  2. **b** is type Vec_d containing a $\mathbb{R}^{n}$ vector defined in (1).
- Main Outputs
  1. **x** is type Vec_d containing a $\mathbb{R}^{d}$ 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 scalar. flag=0 indicates LSQR has converged. flag=1 indicates LSQR has not converged.
  3. **it** is a scalar 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 1.4 chosen by calibration [4].
  2. **k** is the 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 2 chosen by calibration [4].
  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 2. See SuiteSparseQR documentation for more information.
  5. **it_tol** specifies the relative residual tolerance for LSQR convergence. The default value is 10^{-6}.

6. **max_it** specifies the maximum iteration of LSQR, the default value is $10,000$.
7. **rcond_thres** 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^{-10}
8. **perturb** is a real number. When the preconditioner returned by sparse QR is ill conditioned as defined by rcond, any computation with the preconditioner is perturbed (details see our paper) by perturb. The default value is 10^{-10}.
9. **debug** is a flag. If debug=1, the solver prints additional outputs for diagnosis.

Example The following program (`test/ski-llsSparseExample.cpp`) solves an example of problem (1) with given sparse A and dense b, where

$$A = \begin{pmatrix} 2 & 3 & 0 \\ 0 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 5 \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} 0.0649338 \\ 0.845946 \\ -0.0164085 \\ 0.247119 \end{pmatrix}. \tag{3}$$

```
#include <iostream>
#include "ski-lls.h"

int main(int argc, char **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};

    cs_dl* A;
    A = cs_dl_spalloc(m, n, nnz, 1,0 );

    A->p = (long*)col;
    A->i = (long*)row;
    A->x = (double*)val;


    double data_b[4] = {0.0649338,0.845946,-0.0164085,0.247119};
    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 abs_tol = ABS_TOL;
    int ordering = SPQR_ORDERING;
    int debug = 0;

    // storage

    long it;
```

```
    int flag;
    long rank;
    double t_start;
    double t_finish;
    double residual;
    Vec_d x(A->n);

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

    // compute residual
    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, -1, b);

    std::cout << "A is: " << std::endl;
    cs_dl_print(A,0);
    std::cout << "b is: "<< b << std::endl;
    std::cout << "solution is: "<< x << std::endl;
    // Should get      x = (0.0571 -0.0164 0.1127)

    std::cout << "Residual of ls_sparse_spqr is: "<<  nrm2(b) << std::endl;

}
```

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] C. Cartis, J. Fiala, and Z. Shao. Sparse sketching for sparse linear least squares. *ICML Workshop Beyond first order methods for ML systems*, 2020. available at https://drive.google.com/file/d/1BacyZwtZSKZBBblLC7x4SikuDQaLHuYK/view.

[4] C. Cartis, J. Fiala, and Z. Shao. Hashing embeddings of optimal dimension, with applications to linear least squares. *Optimization-Online*, 2021. available at http://www.optimization-online.org/DB_HTML/2021/05/8413.html.

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

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

[7] P.-G. Martinsson, G. Quintana Ortí, N. Heavner, and R. van de Geijn. Householder QR factorization with randomization for column pivoting (HQRRP). *SIAM J. Sci. Comput.*, 39(2):C96–C115, 2017.

[8] 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.

[9] 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.