

# Computing Large Scale Linear Least Squares with SKiLLS

Zhen Shao

January 12, 2021

## 1 Overview

SKiLLS (SKetchIng-Linear-Least-Squares) 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

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

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.

## 2 Use of External Library

The location of external libraries are coded in a config.mk file, given below.

```
1 # External library Locations
2 SUITSPARSEROOT ?= /home/shaoz/SuiteSparse
3 LAPACKROOT ?= /home/shaoz/lapack 3.8.0/
4 FortranROOT ?= /home/shaoz/intel/lib/intel64
5 HSLROOT ?= /home/shaoz/hsledits
6 RCPQR_ROOT ?=/Users/zhen/Downloads/hqrrp/lapack_compatible_sources
7 FFTW_ROOT?= /home/shaoz/usr/lib
8 LIBS_MKL = Wl, start group ${MKLROOT}/lib/intel64/libmkl_intel_ilp64.a \
9   ${MKLROOT}/lib/intel64/libmkl_sequential.a \
10  ${MKLROOT}/lib/intel64/libmkl_core.a Wl, liomp5 lpthread lm ld1
11
12 # Optimization Level
13 OPTIMIZATION = O3
14 CXX=icc
15
16 INCLUDES = I./include I./include/impl I${SUITSPARSEROOT}/include I${BOOSTROOT} \
17   I${FFTW_INCLUDE}
18 CXXFLAGS = Wall ${OPTIMIZATION} march=native fPIC DMKL_ILP64 I${MKLROOT}/include std=c++11
19 CXXFLAGS_DEBUG = Wall O0 g march=native fPIC DMKL_ILP64 I${MKLROOT}/include
```

So that it's easy to see the required dependency for the package, and use the package on different systems (I need to run the software on three different machines.).

### 3 Generating a sparse dimensionality reduction matrix in compressed column format

```
20 // Generate an array 1:n
21 void gen_0_to_n(long n, long *returned_array)
22 {
23     for (long i = 0; i<n; i++){
24         returned_array[i] = i;
25     }
26 }
27
28 // Randomly shuffle an array so that the first k indices are sampled from 1:n without replacement
29 void select_k_from_n(
30     long n /* length of array */,
31     long k /*number of indices to be shuffled*/,
32     long* array_to_be_shuffled)
33 {
34     // input check
35     assert(n >= k);
36     for (long i=0; i<k; i++){
37         long j = ( rand() % (n - i) ) + i;
38         long tmp = array_to_be_shuffled[j];
39         array_to_be_shuffled[j] = array_to_be_shuffled[i];
40         array_to_be_shuffled[i] = tmp;
41     }
42 }
43
44 // Generate a hashing matrix, return in the compressed column sparse format
45 void gen_hashing_matrix(
46     long m /* number of rows */,
47     long n /* number of columns */,
48     long nnz_per_column /* number of non zeros per column */,
49     // output
50     long *row_indices , long *col_array , double *values)
51 {
52     long one_to_m[m];
53     gen_0_to_n(m, one_to_m);
54     for (int i=0; i<n; i++){
55         select_k_from_n(m, nnz_per_column, one_to_m);
56         for (int j=0; j<nnz_per_column; j++){
57             row_indices[nnz_per_column*i + j] = one_to_m[j];
58         }
59     } // the above loop generate the row indices
60
61     for (long i=0; i<n+1; i++){
62         col_array[i] = nnz_per_column*i;
63     } // generate the column array
64
65     double scaling = sqrt(nnz_per_column);
66     for (long i=0; i<n; i++){
67         for (long j =0; j<nnz_per_column; j++){
68             values[i*nnz_per_column +j] = 1/scaling* ((rand()%2)*2-1);
69         }
70     }
71 }
```

## 4 Use abstract linear operator concept for a variety of preconditioner format

```

72 //          SPARSE PRE          //
73 template <typename T>
74 class LinOP_sparse_preconditioner
75     : public LinOp<T>
76 {
77 public:
78     typedef long idx_t;
79     LinOP_sparse_preconditioner( LinOp<T>& A, cholmod_sparse* R,
80     int success=1, double perturb=1e 6)
81         : _A(A) , _R(R), _n(_R>ncol), _m(_A.m()),
82         _success(success), _perturb(perturb)
83     {
84         assert( A.n() == R>nrow );
85         tmp = Vec<T>(A.n());
86     }
87     virtual idx_t const& m() const { return _m;}
88     virtual idx_t const& n() const { return _n;}
89
90     // overwrite matrix vector multiplication by using preconditioner
91     virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
92     {
93         if( trans == 'n' )
94         {
95             tmp = x.copy();
96             cs_utsolve_cholmod_structure(_R, tmp.data(), _success, _perturb);
97             _A.mv( 'n', alpha, tmp, beta, y );
98         }
99         else
100         {
101             _A.mv( 't', 1.0, x, 0.0, tmp );
102             cs_utsolve_cholmod_structure(_R, tmp.data(), _success, _perturb);
103             scal(beta, y);
104             axpy(alpha, tmp, y);
105         }
106     }
107 private:
108     Vec_d      tmp;
109     LinOp<T>&   _A;
110     cholmod_sparse* _R;
111     idx_t _n;
112     idx_t _m;
113     int _success;
114     double _perturb;
115 };
116
117 //          DENSE PRE          //
118 template <typename T>
119 class LinOP_dense_preconditioner
120     : public LinOp<T>
121 {
122 public:
123     typedef long idx_t;
124     LinOP_dense_preconditioner( LinOp<T>& A, Mat_d& R)
125         : _A(A) , _R(R),
126         _up('u'), _trans('t'), _no_trans('n'), _diag('n')

```

```

127 {
128     assert( A.n() == R.m());
129     tmp = Vec<T>(A.n());
130 }
131 virtual idx_t const& m() const { return _A.m();}
132 virtual idx_t const& n() const { return _R.n();}
133
134 // overwrite matrix vector multiplication by using preconditioner
135 virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
136 {
137
138     if( trans == 'n' )
139     {
140         tmp = x.copy();
141         dtrsv(&_up, &_no_trans, &_diag, &_R.n(), _R.data(), &_R.ld(), tmp.data(), &tmp.inc());
142         _A.mv( 'n', alpha, tmp, beta, y );
143     }
144     else
145     {
146         _A.mv( 't', 1.0, x, 0.0, tmp );
147         dtrsv(&_up, &_trans, &_diag, &_R.n(), _R.data(), &_R.ld(), tmp.data(), &tmp.inc());
148         scal(beta, y);
149         axpy(1.0, tmp, y);
150     }
151 }
152 private:
153     Vec_d      tmp;
154     LinOp<T>&   _A;
155     Mat_d&     _R;
156     char       _up;
157     char       _trans;
158     char       _no_trans;
159     char       _diag;
160 };
161
162             IC PRE //
163 template <typename T>
164 class LinOP_ic_preconditioner
165     : public LinOp<T>
166 {
167     public:
168         typedef long idx_t;
169         LinOP_ic_preconditioner( LinOp<T>& A, void* pkeep)
170             : _A(A), _pkeep(pkeep),
171               _trans(1), _no_trans(0), _ifail(0)
172         {
173             // assert( A.n() == R.m()); cannot do the assertion because pkeep is difficult to handle
174             tmp = Vec<T>(A.n());
175         }
176         virtual idx_t const& m() const { return _A.m();}
177         virtual idx_t const& n() const { return _A.n();} // assuming preconditioning size match
178
179         // overwrite matrix vector multiplication by using preconditioner
180         virtual void mv( const char trans, const T alpha, const Vec<T> x, const T beta, Vec<T> y )
181         {
182
183             if( trans == 'n' )
184             {
185                 // Note IC returns Lower triangular, so transpose or no transpose is reversed

```

```

186     hsl_mi35_solve(&_trans, &A.n(), &_pkeep, x.data(), tmp.data(), &_ifail);
187     _A.mv( 'n', alpha, tmp, beta, y );
188 }
189 else
190 {
191     _A.mv( 't', 1.0, x, 0.0, tmp );
192     hsl_mi35_solve(&_no_trans, &A.n(), &_pkeep, tmp.data(), tmp.data(), &_ifail);
193     scal(beta, y);
194     axpy(1.0, tmp, y);
195 }
196 }
197 private:
198     Vec_d      tmp;
199     LinOp<T>&   _A;
200     void*      _pkeep;
201     fint       _trans;
202     fint       _no_trans;
203     fint       _ifail;
204 };
205
206
207 //                                Vanilla LSQR                                //
208
209 void lsqr( LinOp<double>& A, const Vec_d b,
210           const double tol, const long maxit,
211           Vec_d& x, int& flag, long& it, int debug )
212 {
213     assert( A.m() == b.n() );
214     assert( A.n() == x.n() );
215
216     Vec_d v( A.n() );
217     Vec_d w(v.n());
218     double alpha;
219
220     // explicitly make sure it is all zero...
221     for (long i=0; i<A.n(); i++){
222         v(i) = 0;
223     }
224
225     Vec_d u = b.copy();
226     double beta = nrm2(u);
227     if(beta > 0){
228         scal(1.0/beta, u);
229         A.mv( 't', 1.0, u, 0.0, v );
230         alpha = nrm2(v); // norm of v might be zero!!!
231     }
232
233     if (alpha > 0){
234         scal(1.0/alpha, v);
235         w = v.copy();
236     }
237
238     x = 0.0;
239     double nrm_ar = alpha*beta;
240     if (nrm_ar == 0){
241         it=0;
242         flag=0; // converge in 0 iteration
243         return;
244     }

```

```

245
246 double phi, rho;
247 double cs, sn;
248 double phibar = beta;
249 double rhobar = alpha;
250 double theta;
251
252 double nrm_a = 0.0;
253 double nrm_r;
254
255 // double nrm_ar_0 = alpha*beta;
256 flag = 1;
257 it = 0;
258 for( long k = 0; k < maxit; ++k )
259 {
260     it++;
261
262     A.mv( 'n', 1.0, v, alpha, u );
263     beta = nrm2(u);
264
265     if (beta>0){
266         scal(1.0/beta,u);
267         nrm_a = sqrt( nrm_a*nrm_a + alpha*alpha + beta*beta );
268         A.mv( 't', 1.0, u, beta, v );
269         alpha = nrm2(v);
270         if (alpha >0){
271             scal(1.0/alpha,v);
272         }
273     }
274
275     rho = sqrt( rhobar*rhobar + beta*beta );
276     cs = rhobar/rho;
277     sn = beta/rho;
278     theta = sn*alpha;
279     rhobar = cs*alpha;
280     phi = cs*phibar;
281     phibar = sn*phibar;
282
283     axpy( phi/rho, w, x );
284
285     scal( theta/rho, w );
286     axpy( 1.0, v, w );
287
288     nrm_r = phibar;
289     nrm_ar = phibar*alpha*fabs(cs);
290
291     if( nrm_ar < tol*nrm_a*nrm_r ){
292         flag = 0;
293         break;
294     }
295 }
296 }
297 }
298
299 // Preconditioned version, sparse preconditioner
300 void lsqr( LinOp<double>& A, const Vec_d b,
301            const double tol, const long maxit,
302            Vec_d& x, int& flag, long& it, cholmod_sparse* R_11,
303            int success, double perturb, int debug) // LSQR with preconditioner

```

```

304 {
305     LinOP_sparse_preconditioner<double> Apre(A, R_11, success , perturb);
306     lsqr( Apre, b, tol, maxit, x, flag, it, debug );
307 }
308
309
310 //                                     Preconditioned lsqr, dense preconditioner                                     //
311
312 void lsqr_dense_pre( LinOp<double>& A, const Vec_d b,
313                     const double tol, const long maxit,
314                     Vec_d& x, int& flag, long& it, Mat_d& R, int debug) // LSQR with preconditioner
315 {
316     LinOP_dense_preconditioner<double> Apre(A, R);
317     lsqr( Apre, b, tol, maxit, x, flag, it, debug);
318 }
319
320
321                                     Preconditioned lsqr, using incomplete cholesky                                     //
322 void lsqr_ic( LinOp<double>& A, const Vec_d b,
323              const double tol, const long maxit,
324              Vec_d& x, int& flag, long& it, void* pkeep, int debug) // LSQR with ic preconditioner
325 {
326     LinOP_ic_preconditioner<double> Apre(A, pkeep);
327     lsqr( Apre, b, tol, maxit, x, flag, it, debug);
328 }

```

## 5 One version of randomised sparse linear least squares solver

```

329
330 // C++ solver for dense linear least squares using hashing
331 // Solving the linear least squares min_x ||Ax - b||_2
332 void ls_dense_hashing_blendenpik(
333     Mat_d & A, /* m by n */
334     Vec_d& b, /* m by 1 */
335     Vec_d& x, /* solution, n by 1 */
336     long& rank, /* detected rank (in case CPQR is used, otherwise equals to n) */
337     int& flag, /* LSQR convergence flag (0=not convergent, 1=convergent) */
338     long& it, /* LSQR iteration count */
339     double gamma, /* over sampling ratio */
340     long k, /* nnz per column in the hashing matrix */
341     double abs_tol, /* absolute tolerance for the residual */
342     double rcond, /* control minimal diagonal entries of R_11 */
343     double it_tol, /* LSQR relative tolerance */
344     long max_it, /* LSQR max iteration */
345     int debug, /* no use */
346     int wisdom /* flag that if fftw wisdom is to be used */
347 )
348 {
349     assert( A.m() == b.n() );
350     assert( A.n() == x.n() );
351     if (A.m() < A.n()) {
352         throw std::runtime_error( "Matrix_is_not_overdetermined." );
353     }
354     long n = A.n();
355     double t1;
356     double t2;
357     double t_tmp;
358 }

```

```

359 // sketch
360 Vec_d c(ceil(A.n()*gamma)); // RHS to be sketched
361 Mat_d As = rand_hashing_dct(A, b, c, gamma, k, wisdom);
362
363 // build preconditioner, test explicit sketching solution
364 long* E;
365 E = (long*) calloc(As.n(), sizeof(long));
366 double *tau = (double*) calloc(n, sizeof(double));
367 // Compute CPQR factorization //
368 column_pivoted_qr(As, E, tau);
369
370 // Compute Q^T c, store the result in c //
371 char left= 'L';
372 char right = 'R';
373 char trans = 'T';
374 char no_trans = 'N';
375 long one = 1;
376 long two = 2;
377 long workspace_size, info;
378 double *workspace;
379 double wsize_d;
380
381 /* Query workspace size */
382 workspace_size = 1;
383 workspace = &wsize_d;
384 dormqr(&left, &trans, &c.n(), &one, &As.n(), As.data(), &As.ld(), tau, c.data(), &c.n(),
385 workspace, &workspace_size, &info);
386 /* Compute */
387 workspace_size = (long)wsize_d;
388 workspace = (double *)malloc(sizeof(double) * workspace_size);
389 dormqr(&left, &trans, &c.n(), &one, &As.n(), As.data(), &As.ld(), tau, c.data(), &c.n(),
390 workspace, &workspace_size, &info);
391
392 // get the R and the rank //
393 rank=0;
394 for (long i=0; i< As.n(); i++){
395     if (fabs( As(i,i) ) > rcond)
396     {
397         rank++;
398     }
399     else
400     {
401         break;
402     }
403 }
404 Mat_d R;
405 R = As.submat(0, rank, 0, rank);
406
407 // Calculate the least square solution by back substitution /
408 char up = 'u';
409 char diag = 'n';
410 dtrsv(&up, &no_trans, &diag, &R.n(), R.data(), &R.ld(), c.data(), &c.inc());
411
412 // Get the basic solution //
413
414 for (long i=0; i < rank; i++){
415     x.data()[E[i] 1] = c.data()[i];
416 }
417

```



```

418     for (long i = rank; i < A.n(); i++){
419         x.data()[E[i] 1] = 0;
420     }
421     //                               Test residual                               //
422     Vec_d rs = b.copy();
423     A.mv('n', 1, x, 1, rs);
424     std::cout << "Explicit_Sketching_Residual_is:_<< nrm2(rs) << std::endl;
425
426     if (nrm2(rs) < abs_tol){
427         std::cout << "Return_solution_found_by_explicit_sketching_with_residual:_<< nrm2(rs)
428         it = 0;
429         flag = 0;
430         return;
431     }
432     Vec_d xs = x.copy();
433
434     // subselect columns of A
435     long forward = 1;
436     long backward = 0;
437     dlapmt(&forward, &A.m(), &A.n(), A.data(), &A.ld(), E);
438     Mat_d A_reduced;
439     A_reduced = A.submat(0, A.m(), 0, rank);
440     Vec_d y(rank);
441
442     // preconditioned lsqr (dense preconditioner)
443     lsqr_dense_pre(A_reduced, rs, it_tol, max_it, y, flag, it, R, debug);
444     dtrsv(&up, &no_trans, &diag, &R.n(), R.data(), &R.ld(), y.data(), &y.inc());
445     // recover original solution by doing the permutation
446
447     for (long i=0; i < rank; i++){
448         x.data()[E[i] 1] = y.data()[i];
449     }
450
451     for (long i = rank; i < A.n(); i++){
452         x.data()[E[i] 1] = 0;
453     }
454
455     // add the explicit sketching solution xs (initial guess)
456     for (long i=0; i < x.n(); i++){
457         x(i) = xs(i) + x(i);
458     }
459
460     // bring A back to its original form
461     dlapmt(&backward, &A.m(), &A.n(), A.data(), &A.ld(), E);
462     free(E);
463 }

```